ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή Εργασία:

Διαχείριση Φορτίου Σε Κατανεμημένα Συστήματα Αντικειμένων

Επιμέλεια:

ΠΑΝΑΓΙΩΤΗΣ ΓΡΟΝΤΑΣ Π/96020

Επίβλεψη:

ΕΠΙΚΟΥΡΟΣ ΚΑΘΗΓΗΤΗΣ **ΧΡΗΣΤΟΣ ΔΟΥΛΗΓΕΡΗΣ**

Πειραιάς,

Σεπτέμβριος 2000.

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΙΝΑΚΑΣ ΣΧΗΜΑΤΩΝ

1. ΕΙΣΑΓΩΓΗ

1.1 Κατανεμημένα Συστήματα.

Τα κατανεμημένα συστήματα λειτουργούν σε δικτυακά περιβάλλοντα. Αρχικά ως στόχο είχαν να δώσουν στον χρήστη τους την αίσθηση ότι δουλεύει σε ένα μόνο υπολογιστή αποκρύπτωντας του το δίκτυο. Η συγκεκριμένη εργασία ασχολείται με τεχνολογίες οι οποίες έχουν ως σκοπό να αποκρύψουν το δίκτυο και από αυτόν που αναπτύσσει το κατανεμημένο σύστημα. Να του κρύψουν δηλαδή τις λεπτομέρειες της επικοινωνίας μεταξύ των μηχανών, του εντοπισμού των διαφόρων διεργασιών και άλλες *'ευχάριστες'* λεπτομέρειες, κάνοντας την ανάπτυξη ενός κατανεμημένου συστήματος υπόθεση ανάλογη με αυτήν της ανάπτυξης ενός κλασικού προγράμματος.

Αν προσπαθούσαμε να δώσουμε έναν ορισμό για το τι είναι κατανεμημένο σύστημα θα λέγαμε ότι είναι ένα σύστημα που χρησιμοποιεί ένα σύνολο από συνδεδεμένους, αλλά αυτόνομους υπολογιστές για να λύσουν συνολικά κάποιο πρόβλημα. Το κατανεμημένο σύστημα παρουσιάζει το παραπάνω σύνολο υπολογιστών ως ένα στον χρήστη και έτσι η διεπαφή του με αυτόν είναι η ίδια ανεξάρτητα με την φυσική θέση στην οποία αυτός βρίσκεται [17].

Τα κατανεμημένα συστήματα προέκυψαν καθώς ορισμένες εφαρμογές είναι κατανεμημένες από την φύση τους. Αυτό συμβαίνει γιατί [16] :

1. Τα δεδομένα που χρησιμοποιούνται είναι κατανεμημένα δηλαδή βρίσκονται σε διαφορετικές μηχανές, οι οποίες επικοινωνούν μέσω ενός τοπικού ή ευρείας ζώνης δικτύου. Η θέση των δεδομένων δεν μπορεί να αλλάξει για λόγους διαχείρισης ή ιδιοκτησίας (δηλαδή ο ιδιοκτήτης τους μπορεί να επιτρέπει την απομακρυσμένη πρόσβαση αλλά όχι την αντιγραφή).
2. Επειδή είναι δυνατή (από την φύση του προβλήματος, ίσως) η εκμετάλλευση πολλών επεξεργαστών ώστε η επίλυση του προβλήματος να είναι παράλληλη, με καλύτερα αποτελέσματα στον χρόνο εκτέλεσης.
3. Επειδή οι χρήστες της εφαρμογής είναι κατανεμημένοι και επικοινωνούν μεταξύ τους μέσω της εφαρμογής (Για παράδειγμα τα προγράμματα *chat*).

Οι παραπάνω λόγοι όμως δεν θα ήταν αρκετοί για να καθιερώσουν τα κατανεμημένα συστήματα. Το κυριότερο σημείο για την πρόοδο τους είναι αναμφισβήτητα, η άφθονη επεξεργαστική ισχύς που διαθέτουν ακόμα και οι πιο φθηνοί μικροϋπολογιστές που κατασκευάζονται και διατίθενται σήμερα στην αγορά. Επίσης δεν θα έπρεπε να παραλείψει κανείς την πρόοδο στην δικτυακή υποδομή στους λόγους που κάνουν εφικτή, ίσως και επιτακτική, την λειτουργία ενός κατανεμημένου συστήματος.

Οι κατανεμημένες εφαρμογές διαφέρουν σε αρκετά ουσιαστικά σημεία από τις εφαρμογές που εκτελούνται μόνο σε μία μηχανή. Συγκεκριμένα η ταχύτητα επικοινωνίας μεταξύ διεργασιών στην ίδια μηχανή είναι αρκετά μεγαλύτερη από αυτήν των ίδιων διεργασιών, όταν αυτές βρίσκονται σε διαφορετικές μηχανές. Αν η λειτουργία γίνεται μάλιστα πάνω από δίκτυα ευρείας ζώνης κάτι τέτοιο αποκτά ζωτική σημασία. Κατά συνέπεια πρέπει να αποφεύγονται οι εφαρμογές που τα διάφορα συστατικά τους έχουν πολύ μεγάλη σχέση μεταξύ τους (*σύζευξη* κατά την Τεχνολογία Λογισμικού). Επίσης μια πολύ σημαντική διαφορά είναι ότι σε μια κατανεμημένη εφαρμογή οι διάφορες διεργασίες μπορούν να 'αποτύχουν' ξεχωριστά, ενώ στον αντίποδα θα αποτύχουν όλες μαζί και άρα και η εφαρμογή. Κατά συνέπεια πρέπει να μελετηθεί η επίδραση που θα έχει σε όλο σύστημα η κατάρρευση μίας διεργασίας σε μια ομάδα με ισχυρή εξάρτηση. Επίσης σε κατανεμημένες εφαρμογές μπορούμε να έχουμε ταυτόχρονη πρόσβαση σε έναν κοινό πόρο. Βέβαια κάτι τέτοιο συμβαίνει και στις μη κατανεμήμενες εφαρμογές (όχι και τόσο συχνά) αν αυτές αποτελούνται από πολλά *νήματα (threads)*. Δημιουργούνται έτσι αρκετά προβλήματα ανταγωνισμού για την πρόσβαση σε κοινούς πόρους. Τέλος στις μη κατανεμημένες εφαρμογές δεν υπάρχουν ζητήματα ασφάλειας.

Παρατηρούμε λοιπόν πως η ανάπτυξη κατανεμημένων συστημάτων είναι μια αρκετά δύσκολη και απαιτητική εργασία. Αρκεί να σκεφτούμε ότι εκτός από τα προβλήματα που απλώς αναφέραμε νωρίτερα σε ένα κατανεμημένο συστήμα συναντούμε διαφορετικές αρχιτεκτονικές μηχανών και διαφορετικά λειτουργικά συστήματα κάτι που δυσκολεύει περισσότερο την ανάπτυξη τους. Για τον σκοπό αυτό εξ'αρχής υπήρξε αίτημα για την δημιουργία συγκεκριμένων πλαισίων ανάπτυξης κατανεμημένων συστημάτων τα οποία να 'χειρίζονται' τις παραπάνω λεπτομέρειες, αποκρύπτοντας τις από τους σχεδιαστές ενός κατανεμημένου συστήματος, έτσι ώστε οι τελευταίοι να μπορούν να επικεντρωθούν στην επίλυση του προβλήματος που το συγκεκριμένο σύστημα καλείται να λύσει. Στο παραπάνω αίτημα υπήρξε απόκριση με την μορφή του **DCE (Distributed Computer Environment),** του **CORBA (Common Object Request Broker Architecture)** αλλά και άλλων τέτοιων συστημάτων.

Η σύγχρονη αντιμετώπιση στα κατανεμημένα συστήματα εφαρμόζει τεχνικές της Αντικειμενοστρεφούς Τεχνολογίας για την σχεδίαση και ανάπτυξη τους. Έτσι αναφερόμαστε πλέον σε **Κατανεμημένα Συστήματα Αντικειμένων** (*Distributed Object Systems*). Η αφαίρεση (abstraction) και η τμηματικότητα (modularity), δύο από τα πιο σημαντικά χαρακτηριστικά της Αντικειμενοστρεφούς Τεχνολογίας, ταιριάζουν απόλυτα με τις ανάγκες ενός κατανεμημένου συστήματος, καθώς και στην περίπτωση κατανεμημένου συστήματος και στην περίπτωση τοπικού, το όλο σύστημα μοντελοποιείται ως ένα σύνολο από αντικείμενα που αλληλεπιδρούν μεταξύ τους με μηνύματα (*κλήσεις μεθόδων*). Σε ότι αφορά τον σχεδιασμό, στα κατανεμημένα συστήματα (αντικειμένων) χρησιμοποιείται κυρίως η

αρχιτεκτονική *πελάτη – εξυπηρετητή (client - server)*, η οποία μοντελοποιεί την αλληλεπίδραση δύο οντοτήτων για την επίλυση ενός προβλήματος, ως μια σειρά από αιτήσεις και απαντήσεις στις αιτήσεις. Η συγκεκριμένη αρχιτεκτονική έχει δύο χαρακτηριστικά τα οποία την κάνουν ιδανική για τέτοιες περιπτώσεις. Αυτά είναι η *επεκτασιμότητα (scalability)* και η *τμηματικότητα (modularity)*. Σε ότι αφορά το δεύτερο τα συστατικά μιας εφαρμογής χωρίζονται σε αυτά που παρέχουν μια υπηρεσία (εξυπηρετητές) και σε αυτά που ζητούν και χρησιμοποιούν για δικό τους όφελος μια υπηρεσία (πελάτες). Έτσι έχουμε την κατανομή της επεξεργασίας μεταξύ των δύο παραπάνω οντοτήτων. Η επεκτασιμότητα προκύπτει άμεσα και με πολύ φυσικό τρόπο, καθώς ένα αντικείμενο που είναι πελάτης προς ένα άλλο, μπορεί να είναι εξυπηρετητής προς ένα τρίτο κ.ό.κ.

1.2 Διαχείριση Φορτίου

Ένα από τα πιο σημαντικά ζητήματα στην κατασκευή κατανεμημένων συστημάτων είναι η κατανομή των διαθέσιμων πόρων. Τα διάφορα αιτήματα για επεξεργασία καταφθάνουν στο σύστημα με τυχαίο τρόπο, με συνέπεια την άνιση κατανομή τους στους κόμβους επεξεργασίας. Η παραπάνω κατάσταση έχει αντίκτυπο και σε όλα τα υπόλοιπα υποσυστήματα, με αποτέλεσμα ορισμένοι ‘κόμβοι’ να είναι φορτωμένοι ενώ άλλοι να μην χρησιμοποιούν το πλήρες φάσμα των δυνατοτήτων τους. Το παραπάνω οδηγεί στην μείωση της συνολικής απόδοσης του συστήματος. Είναι απαραίτητη λοιπόν η εξάλειψη αυτής της ‘τυχαιότητας’ που διαδίδεται στο όλο σύστημα, με κατανομή των εισερχόμενων εργασιών με τρόπο ώστε να υπάρχει ισορροπία στο φορτίο του κάθε κόμβου, με απώτερο στόχο την βελτίωση της απόδοσης του συστήματος και της δυνατότητας να παρέχει τις υπηρεσίες για τις οποίες σχεδιάστηκε.

Αν προσπαθούσαμε να εκφράσουμε με τρόπο γενικό το πρόβλημα της διαχείρισης φορτίου, θα λέγαμε [8] ότι είναι η ανάθεση συγκεκριμένων οντοτήτων σε προκαθορισμένους στόχους έτσι ώστε να ικανοποιούνται ένα σύνολο από απαιτήσεις που συνήθως αφορούν θέματα απόδοσης. Σε ένα περιβάλλον client – server, οι οντότητες που κατανέμονται είναι οι αιτήσεις για επεξεργασία και οι στόχοι είναι οι διαθέσιμοι εξυπηρετητές. Οι απαιτήσεις απόδοσης που πρέπει να ικανοποιηθούν για τους clients είναι η ελαχιστοποίηση του χρόνου στον οποίο λαμβάνουν απόκριση από το σύστημα, ενώ για τους servers η μεγιστοποίηση της απόδοσης, δηλαδή του αριθμού των αιτήσεων που επεξεργάζονται σε ένα συγκεκριμένο χρονικό διάστημα.

Το μεγαλύτερο ίσως πεδίο εφαρμογής των παραπάνω είναι το περιβάλλον του *Παγκόσμιου Ιστού (WWW – World Wide Web)*, που ως γνωστόν χρησιμοποιεί το μοντέλο client-server. Οι πελάτες είναι οι web browsers, οι οποίοι στέλνουν αιτήσεις στους web servers και οι οποίοι απαντούν στέλνοντας σελίδες HTML *(HyperText Markup Language)*, αλλά και άλλου τύπου

περιεχόμενο, πχ. πολυμεσικό. Η όλη επικοινωνία γίνεται χρησιμοποιώντας το πρωτόκολλο HTTP *(HyperText Transfer Protocol)*. Η τεράστια απήχηση του WWW έχει δημιουργήσει αρκετά προβλήματα φορτίου στα διάφορα sites, τα δημοφιλέστερα από τα οποία δέχονται εκατομμύρια επισκέψεις καθημερινά. Κατά συνέπεια, οι web servers πρέπει να χειρίζονται τεράστιο αριθμό αιτήσεων, κάτι που όπως είναι φανερό δημιουργεί τεράστιο φορτίο. Από νωρίς λοιπόν ([6] - 1994) έχουν προταθεί τεχνικές για την καλύτερη κατανομή των αιτήσεων που γίνονται σε ένα web site. Το κοινό σημείο των τεχνικών αυτών είναι ότι διατηρούν πολλούς web servers για κάθε site, στους οποίους διανέμουν με κάποιον τρόπο τις διάφορες αιτήσεις, ώστε να εξισορροπούν κάπως το φορτίο. Οι τρόποι διανομής που έχουν προταθεί είναι είτε ο *Round-Robin*, είτε ο χωρισμός των διαφόρων ιστοσελίδων ανάλογα με το περιεχόμενο τους, ώστε κάθε αίτηση για σελίδα να εξυπηρετείται ανάλογα με τον web server που την περιέχει. Το κοινό στοιχείο όλων των υλοποιήσεων του πρώτου τρόπου διανομής είναι ότι η κατανομή του φορτίου γίνεται στο επίπεδο δικτύου και είναι συνήθως ενσωματωμένο στον μεταγωγέα με τον οποίο το site συνδέεται με τον έξω κόσμο.

Οι παραπάνω μέθοδοι δουλεύουν ικανοποιητικά όταν πρόκειται για HTML σελίδες. Όμως το web εξελίσσεται και πολλές φορές ξεφεύγει από το μοντέλο που περιγράψαμε νωρίτερα. Το περιβάλλον HTTP χρησιμεύει μόνο ως το αρχικό πλαίσιο για την εκτέλεση περίπλοκων εφαρμογών (*Object Web*). Αναπτύσσονται πλέον διάφορες εφαρμογές που ξεφεύγουν από το γνωστό σενάριο ‘αίτηση σελίδας – παροχή σελίδας’. Κατά συνέπεια και οι μέθοδοι για διαχείριση του φορτίου γίνονται πιο δύσκολα εφαρμόσιμες. Το πιο σημαντικό πρόβλημα που αντιμετωπίζουν αυτές είναι ότι δουλεύουν στο επίπεδο δικτύου. Κατά συνέπεια ‘βλέπουν’ μόνο πακέτα δεδομένων, από τα οποία μπορούν να εξαγάγουν ελάχιστες πληροφορίες όπως για παράδειγμα τον host από τον οποίο προέρχονται και τον host στον οποίο προορίζονται. Δεν μπορούν για παράδειγμα να γνωρίζουν σε ποια εφαρμογή ‘ανήκουν’ αυτά, ή ποια είναι τα χαρακτηριστικά του χρήστη της συγκεκριμένης εφαρμογής. Μία λύση σε αυτό το πρόβλημα είναι η ανάθεση της λειτουργίας της κατανομής του φορτίου σε ένα ανώτερο επίπεδο, για παράδειγμα, στο επίπεδο εφαρμογής. Εδώ τα ταυτόσημα πακέτα αποκτούν κάποιο νόημα και η παρακολούθηση στοιχείων όπως αυτά που προαναφέραμε γίνεται πιο εύκολη.

Η Αντικειμενοστρεφής Τεχνολογία μπορεί να βοηθήσει και σε αυτόν τον τομέα. Τα αντικείμενα προβάλλουν ως το ελάχιστο στοιχείο μιας εφαρμογής στο οποίο μπορεί να γίνει κατανομή με σκοπό την διαχείριση του φορτίου. Η συγκεκριμένη εργασία ασχολείται με την κατανομή αντικειμένων σε ένα περιβάλλον επιπέδου εφαρμογής, το οποίο μπορεί να ονομαστεί ‘επίπεδο αντικειμένου’. Γίνεται προσπάθεια να εφαρμοστούν ορισμένες από τις τεχνικές που χρησιμοποιούνται για την κατανομή του φορτίου σε περιβάλλονται HTML, σε ένα περιβάλλον που λειτουργεί χρησιμοποιώντας αντικείμενα.

Η διάρθρωση της συγκεκριμένης εργασίας έχει ως εξής:

- ⇒ Στο *Κεφάλαιο 2* περιγράφονται αναλυτικά τα διάφορα μοντέλα που μπορούν να χρησιμοποιηθούν για την κατασκευή κατανεμημένων συστημάτων αντικειμένων.
- ⇒ Στο *Κεφάλαιο 3* ορίζονται κάποιες γενικές έννοιες σχετικά με την διαχείριση φορτίου και περιγράφονται ορισμένες τεχνικές που εφαρμόζονται για τον σκοπό αυτό σε περιβάλλον WWW.
- ⇒ Στο *Κεφάλαιο 4* περιγράφεται η λειτουργία και η αρχιτεκτονική του συστήματος που αναπτύσσει η συγκεκριμένη εργασία.
- ⇒ Στο *Κεφάλαιο 5* δίνονται κάποια συμπεράσματα που έχουν ληφθεί από την λειτουργία του συστήματος καθώς και κάποιες πιθανές επεκτάσεις του.
- ⇒ Τέλος, στο Παράρτημα δίνεται και περιγράφεται ο πηγαίος κώδικας του υπό ανάπτυξη συστήματος, καθώς επίσης και ορισμένες γενικές οδηγίες για την λειτουργία του.

2. ΤΕΧΝΟΛΟΓΙΕΣ

Στην παρούσα ενότητα θα αναφερθούμε στις τεχνολογίες πάνω στις οποίες βασίζεται το σύστημα που υλοποιεί η συγκεκριμένη εργασία. Αρχικά θα γίνει αναλυτική περιγραφή κάθε μιας από αυτές . Στο τέλος θα υπάρξει μία ενσωμάτωση τους στην οποία θα περιγραφεί το περιβάλλον εκτέλεσης του συστήματος που αναπτύσσεται.

Σε ότι αφορά τα κατανεμημένα συστήματα είναι γεγονός πως αυτά μπορούν να αναπτυχθούν χωρίς να υπάρχει ανάγκη από κάποιο πλαίσιο, ή συγκεκριμένη τεχνολογία, όπως το RMI και το CORBA. Η διαδικασία αυτή μπορεί να γίνει με τα πρωτογενή δικτυακά στοιχεία που παρέχει κάθε περιβάλλον προγραμματισμού. Ένα από αυτά είναι η πρόσβαση και η χρήση των sockets. Ο συγκεκριμένος τρόπος ανάπτυξης όμως μπορεί εύκολα να παραλληλιστεί με τον προγραμματισμό σε *assembly*. Κατά συνέπεια, γίνεται φανερή η ανάγκη για συστήματα τα οποία θα 'αποκρύπτουν' κατά κάποιον τρόπο τόσο 'τεχνικές' λεπτομέρειες και βεβαίως θα είναι δοκιμασμένα και κυρίως έτοιμα για σημαντικές εφαρμογές, συστήματα δηλαδή τα οποία θα παρέχουν ένα μεγαλύτερο επίπεδο αφαίρεσης. Το πρώτο τέτοιο σύστημα ήταν το **RPC (Remote Procedure Call)** το οποίο επέτρεπε την εκτέλεση διαδικασιών από έναν υπολογιστή σε κάποιον άλλον. Βέβαια το RPC δεν ήταν αντικειμενοστρεφές, καθώς επέτρεπε μόνο την χρήση πρωτογενών τύπων δεδομένων ως ορίσματα και χρησιμοποιούσε διαδικασίες και όχι μεθόδους[1]. Ήταν και είναι όμως ο πρόγονος όλων των κατανεμημένων συστημάτων που παρουσιάζονται στην συνέχεια. Πριν την περιγραφή τους όμως, θα γίνει μια μικρή αναφορά στην γλώσσα υλοποίησης της συγκεκριμένης εργασίας, την **Java.**

2.1 Java

Η Java είναι η πιο νέα γλώσσα προγραμματισμού ευρείας χρήσης. Αναπτύχθηκε από τον James Gosling και την ομάδα του στα εργαστήρια της Sun Microsystems το 1995. Έχει χαρακτηριστεί με επιτυχία ως ένα 'λειτουργικό συστημα αντικειμένων' [2, 3, 4]. Η Java θα ήταν μια κλασσική αντικειμενοστρεφής γλώσσα προγραμματισμού σαν την C++ , αν δεν είχε τα εξης δύο μοναδικά χαρακτηριστικά:

- *Μεταφερσιμότητα (Run - Time - Portability):* Θεωρητικά τα προγράμματα Java μπορούν να τρέξουν σε διαφορετικές πλατφόρμες χωρίς να υπάρξει ανάγκη για επανα-μεταγλώττιση τους. Αυτό επιτυγχάνεται με αρχική μεταγλώττιση του πηγαίου κώδικα σε μια ενδιάμεση, ανεξάρτητη μορφή που ονομάζεται *bytecodes*. Η

[1] Μεταξύ των εννοιών διαδικασία και μέθοδος υπάρχει η εξής διαφορά: Μια διαδικασία κάθε φορά που εκτελείται θα έχει τα ίδια αποτελέσματα. Αντίθετα τα αποτελέσματα μιας μέθοδου εξαρτώνται άμεσα από την κατάσταση του αντικειμένου στο οποίο καλείται. Έτσι μια μέθοδος των αντικειμένων Α και Β που ανήκουν στην ίδια κλάση θα έχει πιθανότατα διαφορετικά αποτελέσματα όταν εκτελεστεί.

ενδιάμεση αυτή μορφή δεν εκτελείται άμεσα από το υλικό. Αντίθετα φορτώνεται και εκτελείται από έναν διερμηνέα λογισμικού (interpreter) στον οποίο έχει δοθεί το πολλά υποσχόμενο όνομα *Java Virtual Machine (JVM)*. Ο διερμηνέας αυτός έχει γραφτεί ειδικά για την πλατφόρμα εκτέλεσης, άλλος δηλαδή για Windows, άλλος για Unix κτλ. Ο διερμηνέας αυτός έχει ως ευθύνη να φορτώνει δυναμικά τα bytecodes και να τα εκτελεί. Η μεταφερσιμότα που επιτυγχάνεται με τον παραπάνω τρόπο έχει ως αρνητική συνέπεια την μείωση της ταχύτητας εκτέλεσης των προγραμμάτων. Αρχικά η μείωση αυτή ήταν πολύ σημαντική σε σχέση με την ταχύτητα μιας γλώσσας όπως η C. Σιγά, σιγά υπήρξε κάποια βελτίωση χωρίς όμως να υπάρξει ποτέ εξισορρόπηση.

- *Mobile Code:* Το χαρακτηριστικό της δυναμικής φόρτωσης και εκτέλεσης κώδικα από μία JVM δίνει την εξής σημαντική δυνατότητα στα συστήματα Java. Τα bytecodes μπορούν να τοποθετούνται σε μία μόνο φυσική τοποθεσία και να φορτώνονται κατ'απαίτηση όποτε αυτά είναι απαραίτητα (*on demand*). Η διανομή των εφαρμογών γίνεται έτσι εντελώς αυτόματη. Το χαρακτηριστικό αυτό εφαρμόστηκε με επιτυχία όταν η Netscape ενσωμάτωσε σε κάθε Communicator μια JVM. Τα bytecodes τοποθετούνταν στον web server και φορτώνονταν με χρήση του <*APPLET*> tag από την σελίδα HTML. Έτσι δημιουργήθηκαν τα περίφημα applets. Διαφωνίες βέβαια της Sun με την Netscape και την Microsoft σχετικά με το ποιά έκδοση της JVM θα μπει στους browser περιόρισαν την χρήση της Java σε περιβαλλον WWW, καθώς οι υπάρχουσες JVMs είναι πλέον ξεπερασμένες και για να επωφεληθούν οι προγραμματιστές από καινούρια χαρακτηριστικά της γλώσσας πρέπει να αναγκάζουν τον κάθε client να κατεβάζει ένα Plug - In για χρήση της πιο πρόσφατης JVM της Sun - μια αρκετά χρονοβόρα διαδικασία, ακατάλληλη για αργές dialup συνδέσεις. Παρ'όλα αυτά η Java παραμένει η καλύτερη λύση για σημαντικές client – server εφαρμογές σε περιβάλλοντα EWB .

Επιπλέον σημαντικά χαρακτηριστικά της Java, τα οποία έχουν παίξει σημαντικό ρόλο στην ανάπτυξη της, είναι το πλούσιο σύνολο βιβλιοθηκών που περιέχει, η ασφάλεια στην εκτέλεση του mobile code και ο πολύ καλός σχεδιασμός της, ο οποίος παρέχει ένα πολύ υψηλό επίπεδο αφαίρεσης στον προγραμματιστή.

2.2 RMI

2.2.1 Γενικά

Το RMI (Remote Method Invocation) είναι μια τεχνολογία, η οποία επιτρέπει την κατασκευή κατανεμημένων συστημάτων αντικειμένων χρησιμοποιώντας μόνο την γλώσσα

Java. Με το RMI τα διάφορα προγράμματα μπορούν να δημιουργούν αντικείμενα τα οποία να μπορούν να δέχονται κλήσεις μεθόδων από άλλα αντικείμενα τα οποία βρίσκονται σε διαφορετικούς hosts. Για να γίνει αυτό τα διάφορα αντικείμενα πρέπει να ξέρουν πώς θα επικοινωνήσουν. Εδώ είναι που παρεμβαίνει το RMI χρησιμοποιώντας ειδικά αντικείμενα τα οποία παράγονται αυτόματα και επιτελούν αυτήν ακριβώς την λειτουργία. Η αρχιτεκτονική του επιτρέπει την εύκολη υλοποίηση κατανεμημένων εφαρμογών καθώς η επικοινωνία μεταξύ δύο αντικειμένων που (ενδεχομένως) βρίσκονται σε διαφορετικές (JV)Μηχανές παρουσιάζεται στον προγραμματιστή ως μια απλή κλήση μεθόδου, κάτι απόλυτα φυσικό δηλαδή, αφού η ίδια τεχνική χρησιμοποιείται και για επικοινωνία με τοπικά αντικείμενα.

Το RMI πρωτοπαρουσιάστηκε το 1997 με το **JDK1.1** και είναι μέρος των βασικών κλάσεων (core) της Java. Κατα συνέπεια σήμερα βρίσκεται παρόν σε όλες τις JVM που κυκλοφορούν στο εμπόριο.

Οι στόχοι του RMI για την κατασκευή κατανεμημένων συστήματων αντικειμένων στην Java είναι [15] :

1. Να επιτρέψει την όσο το δυνατόν ‘φυσικότερη’ ενσωμάτωση δυνατότητας κλήσεων μεθόδων σε απομακρυσμένα αντικείμενα.
2. Να επιτρέψει την δυνατότητα για callbacks από server objects σε applets.
3. Να ενσωματώσει ένα κατανεμημένο μοντέλο αντικειμένων στην Java διατηρώντας την σημασιολογία του τοπικού μοντέλου αντικειμένων.
4. Να παρέχει δυνατότητα για συγγραφή αξιόπιστων κατανεμημένων εφαρμογών.
5. Να διατηρήσει το ασφαλές περιβάλλον της Java.

2.2.2 Αρχιτεκτονική του RMI.

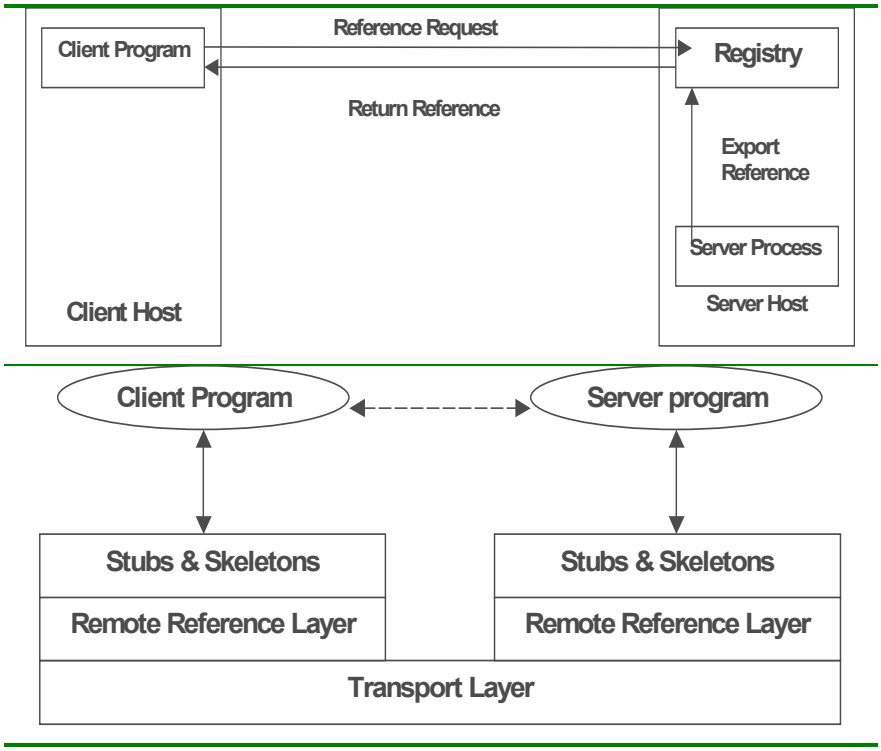
Το RMI είναι ο άμεσος απόγονος του RPC. Επιτρέπει δηλαδή την κλήση μεθόδων με ορίσματα όχι μόνο πρωτογενείς τύπους δεδομένων αλλά και αντικείμενα. Το RMI ακολουθεί το μοντέλο πελάτη - εξυπηρετητή με τα πλεονεκτήματα που περιγράψαμε νωρίτερα. Ο ‘εξυπηρετητής’ είναι ένα αντικείμενο το οποίο παρέχει κάποια υπηρεσία και ο πελάτης το αντικείμενο που θέλει να χρησιμοποιήσει την συγκεκριμένη υπηρεσία. Το RMI βρίσκεται ‘στην μέση’ και παρέχει τον μηχανισμό με τον οποίο τα δύο αντικείμενα αυτά επικοινωνούν, όπως επίσης και διευκολύνει την αλληλεπιδρασή τους.

Συγκεκριμένα, κάθε σύστημα RMI αποτελείται από τουλάχιστον δύο προγράμματα. Το ένα, είναι αυτό που δημιουργεί τα αντικείμενα - εξυπηρετητές και δηλώνει την ύπαρξη τους σε μια υπηρεσία καταλόγου του περιβάλλοντος του RMI (Naming Service). Η υπηρεσία αυτή που ονομάζεται *registry,* στην ορολογία του RMI, στην ουσία αντιστοιχίζει ‘λογικά ονομάτα’ με *object references*. Μια object reference χρησιμοποιείται από το αντικείμενο client για την κλήση της μεθόδου. Με την δήλωση τους στο *registry* λοιπόν, τα αντικείμενα αποκτούν ένα

‘όνομα’. Αυτό το όνομα θα χρησιμοποιούν όσα αντικείμενα θέλουν καλέσουν κάποια μέθοδο των εξυπηρετητών. Από το σχήμα 1, φαίνεται ότι η Naming Service πρέπει να βρίσκεται στον ίδιο υπολογιστή με το πρόγραμμα server. Επίσης πρέπει να σημειωθεί ότι αν τερματίσει η διεργασία του registry χάνονται όλες οι υπάρχουσες αντιστοιχίσεις αντικειμένων και ονομάτων.

Το δεύτερο πρόγραμμα σε ένα περιβάλλον RMI ταυτίζεται με τον πελάτη του μοντέλου και αναζητά στην Naming Service τα αντικείμενα τα οποία θα του παρέχουν την υπηρεσία, χρησιμοποιώντας το όνομα τους. Αποκτά μια reference σε αυτά την οποία και χρησιμοποιεί για να καλέσει μεθόδους.

Στο παρακάτω σχήμα φαίνεται η αρχιτεκτονική ενός συστήματος RMI:



Σχήμα 1. Αρχιτεκτονική συστήματος RMI.

Το πρόγραμμα client είναι αυτό που θέλει να καλέσει μία μέθοδο σε ένα απομακρυσμένο αντικείμενο. Για τον σκοπό αυτό επικοινωνεί με ένα **stub,** δηλαδή με έναν *τοπικό αντιπρόσωπο* του απομακρυσμένου αντικειμένου και καλεί την μέθοδο στο stub. Το stub ‘ξέρει’ πως να βρει το απομακρυσμένο αντικείμενο και να του μεταβιβάσει την κλήση. Η επικοινωνία στην συγκεκριμένη περίπτωση γίνεται μέσω του πρωτοκόλλου *JRMP (Java Remote Method Protocol)* ένα στοιχειώδες connection – oriented πρωτόκολλο. Το stub παρέχει το ίδιο σύνολο υπηρεσιών με το αντικείμενο που αντιπροσωπεύει. Συγκεκριμένα, παρέχει τον ορισμό των υπηρεσιών και όχι την υλοποίηση κάτι που αφήνεται για το αντικείμενο - εξυπηρετητή. Έτσι κι αλλιώς είναι βασική αρχή της Αντικειμενοστρεφούς

Τεχνολογίας ότι ο χρήστης μιας υπηρεσίας δεν ενδιαφέρεται για την υλοποίηση της παρά μόνο για τον ορισμό της. Η αρχή αυτή βρίσκει εφαρμογή στο RMI.

Έχοντας πρόσβαση στο remote αντικείμενο, μέσω του τοπικού αντιπροσώπου του, ο πελάτης μπορεί να καλεί μεθόδους σε αυτό. Οι μέθοδοι που θα κληθούν μπορούν να έχουν ορίσματα κανονικά όπως και σε κάθε 'τοπική' μέθοδο. Το ίδιο ισχύει και για τις επιστρεφόμενες τιμές. Σε ό,τι αφορά το πέρασμα των αντικειμένων από την μία μηχανή στην άλλη (μια διαδικασία που αναφέρεται συχνά ως *marshalling*) ο ακριβής τρόπος υλοποίησης εξαρτάται από τον τύπο του αντικειμένου. Διακρίνουμε 3 περιπτώσεις:

1. Πρωτογενείς τύποι δεδομένων (int , float, κτλ.): Στην περίπτωση αυτή έχουμε πέρασμα by value δηλαδή δημιουργείται ένα αντίγραφο της μεταβλητής και στέλνεται στην απομακρυσμένη μηχανή.

2. Αντικείμενα: Εδώ η υλοποίηση διαφέρει από την μέθοδο που χρησιμοποιείται κανονικά στην Java (pass-by-reference). Το RMI στέλνει ένα αντίγραφο του ίδιου του αντικειμένου στην απομακρυσμένη JVM και όχι ένα αντίγραφο της reference του. Άλλωστε η reference ενός αντικειμένου έχει νόημα μόνο σε έναν υπολογιστή. Παρ'όλα αυτά το πέρασμα της κατάστασης ενός αντικειμένου δεν είναι απλή υπόθεση, καθώς αυτό μπορεί να περιέχει αναφορές σε άλλα αντικείμενα, τα οποία με την σειρά τους να αναφέρονται σε άλλα κόκ . Το RMI χρησιμοποιεί την τεχνική *Java Serialization* για να μετατρέψει ένα αντικείμενο και όλα όσα συνδέονται με αυτό σε μια σειρά από bytes που να μπορούν να αποσταλούν μέσω ενός δικτύου.

3. Απομακρυσμένα αντικείμενα : Το RMI επιτρέπει ένα απομακρυσμένο αντικείμενο να είναι αποτέλεσμα ή όρισμα μιας μεθόδου. Στην περίπτωση αυτή το αντικείμενο δεν στέλνεται όπως περιγράφτηκε προηγουμένως, αλλά αντικαθίσταται και στέλνεται ο τοπικός αντιπρόσωπος του.

Ο 'σημασιολογικός πλούτος' είναι ένα από τα πλεονεκτήματα του RMI έναντι του CORBA, καθώς το τελευταίο επιτρέπει μόνο το πέρασμα πρωτογενών αντικειμένων by - value. Όλοι οι υπόλοιποι τύποι περνούν by reference.

Συνεχίζοντας την περιγραφή του σχήματος 1, παρατηρούμε ότι το ανάλογο του επιπέδου του stub για τον server είναι το επίπεδο skeleton. Το επίπεδο skeleton 'γνωρίζει' πως να μεταβιβάσει τις κλήσεις που θα λάβει από τον πελάτη στο ίδιο το αντικείμενο εξυπηρετητή ώστε να εκτελεστούν οι κατάλληλες μέθοδοι. Επίσης μπορεί να επιστρέψει τυχόν αποτελέσματα των remote μεθόδων. Σε όλες τις εκδόσεις της Java2 ο ρόλος του skeleton έχει αντικατασταθεί πλήρως από το stub και στην μεριά του server. Διατηρείται πλέον μόνο για λόγους συμβατότητας με παλαιότερες υλοποιήσεις.

Σε κάθε περίπτωση η δημιουργία του stub και του skeleton δεν είναι ευθύνη του προγραμματιστή. Σε όλες τις αρχιτεκτονικές που μελετά η συγκεκριμένη εργασία παράγονται αυτόματα από κάποιο εργαλείο που παρέχει το πακέτο που υλοποιεί την αρχιτεκτονική. Η

ενσωμάτωση στα προγράμματα γίνεται με χρήστη των interfaces των αντικειμένων που αντιπροσωπεύουν αυτά.

Προχωρώντας ένα επίπεδο πιο κάτω έχουμε το επίπεδο *Remote Reference*. Υλοποιεί την πραγματική κλήση μεθόδου προς την άλλη πλευρά. Συγκεκριμένα όταν ο client καλεί μια μέθοδο στο stub, το stub απλώς προωθεί την κλήση στο επίπεδο αυτό, το οποίο με την σειρά του την μεταφέρει στο δίκτυο. Από την άλλη πλευρά, η λήψη της κλήσης αναλαμβάνει την ενεργοποίηση του αντικειμένου εξυπηρετητή. Στην αρχική έκδοση του RMI δεν υπήρχε δυνατότητα για αυτόματη ενεργοποίηση των αντικειμένων αυτών. Έπρεπε οπωσδήποτε να υπηρχε κάποιο πρόγραμμα, το οποίο αναλάμβανε να τα αρχικοποιήσει. Το μειονέκτημα αυτό εξαλείφθηκε στην Java2 με την βοήθεια της υπηρεσίας *RMI – Daemon*. Η συγκεκριμένη εργασία χρησιμοποιεί την δυνατότητα αυτόματης ενεργοποίησης των αντικείμενων server.

Τέλος στο επίπεδο μεταφοράς υπάρχει , όπως προαναφέρθηκε το πρωτόκολλο, *JRMP* το οποίο βασίζεται στο TCP. Αυτό βέβαια ισχύει για την υλοποίηση του RMI από την Sun Microsystems. Άλλες υλοποιήσεις έχουν δικά τους πρωτόκολλα και όπως είναι φυσικό, υπάρχει ασυμβατότητα. Το μεγάλο αυτό μειονέκτημα λύθηκε με την υλοποίηση του RMI πάνω από το IIOP (το πρωτόκολλο που χρησιμοποιεί το CORBA στην τωρινή του έκδοση), το οποίο και κυκλοφόρησε επίσημα τον Μάιο του 2000 με το *J2SE1.3*.

Ένα άλλο χαρακτηριστικό του RMI είναι ότι επεκτείνει την αυτόματη διαχείριση μνήμης της Java και για κατανεμημένες εφαρμογές. Όπως είναι γνωστό στην Java δεν υπάρχει ανάγκη για ρητή ανάκτηση της μνήμης που χρησιμοποιούσαν αντικείμενα που δεν χρησιμοποιούνται πλέον. Μια διεργασία γνωστή και ως Garbage Collector (GC) το αναλαμβάνει αυτό. Στο RMI έχουμε τον Distributed Garbage Collector (DGC) ο οποίος αναλαμβάνει να ανακτήσει την μνήμη είτε από αντικείμενα που έχουν χρησιμοποιηθεί αλλά πλέον δεν χρειάζονται, είτε από αντικείμενα που δεν είναι ενεργά πλέον εξ'αιτίας προβλήματος του δικτύου. Ο DGC λειτουργεί σε συνεργασία με τον τοπικό GC και σε γενικές γραμμές χρησιμοποιεί τις εξής δύο τεχνικές:

1. Κάθε φορά που κάποιος client αποκτά μια reference σε ένα remote object, αυξάνεται ένας μετρητής σε αυτό. Όταν σταματά η χρήση του αντικειμένου τότε ο μετρητής αυτός μειώνεται κατά ένα. Όταν φθάσουμε στο μηδέν η μνήμη που έχει κατανεμηθεί στο συγκεκριμένο αντικείμενο μπορεί να ανακτηθεί. Αυτό βέβαια δεν θα γίνει αμέσως, αλλά το συγκεκριμένο αντικείμενο καθίσταται υποψήφιο για αποκομιδή.
2. Επίσης σε κάθε client ο οποίος κάνει σύνδεση με ένα remote αντικείμενο δίνεται μία χρονική περίοδος μέσα στην οποία πρέπει να ανανεώσει την σύνδεση (lease period). Αν σε αυτόν τον χρόνο δεν υπάρξει μια νέα κλήση μεθόδου, τότε η σύνδεση θεωρείται 'χαμένη' λόγω ίσως κάποιου προβλήματος στο δίκτυο.

Τέλος, αξίζει να σημειωθεί ότι στην περίπτωση που το RMI χρειαστεί να μεταφέρει ένα remote αντικείμενο από μία JVM σε μία άλλη, μεταφέρει μόνο την κατάσταση του

αντικειμένου και όχι όλα τα bytecodes της κλάσης. Για τον λόγο αυτόν, το συγκεκριμένο αντικείμενο δεν μπορεί να χρησιμοποιηθεί πρακτικά, αν δεν υπάρχουν τα bytecodes κάπου, όπου μπορεί να τα βρει ο class loader του client. Μία λύση που χρησιμοποιείται σε αυτό το πρόβλημα είναι η χρήση ενός HTTP server. Συγκεκριμένα αν, για παράδειγμα, ένα αντικείμενο-εξυπηρετητής επιστρέφει από μια μέθοδο ένα remote object ενσωματώνει μέσα στο stream που στέλνει στον client το URL, όπου βρίσκονται τα bytecodes. Αυτό το URL δίνεται μέσω της ιδιότητας *java.rmi.server.codebase* κατά την έναρξη της JVM του συγκεκριμένου εξυπηρετητή. Ο client προσπαθεί να φορτώσει τα bytecodes αρχικά τοπικά και αν δεν τα καταφέρει χρησιμοποιεί το URL αυτό. Η τεχνική αυτή δεν αφορά μόνο το κατέβασμα των τοπικών αντιπροσώπων αντικειμένων αλλά μπορεί να χρησιμοποιηθεί και για δυναμικό κατέβασμα όλων των κλάσεων που χρησιμοποιούνται από κάποιο αντικείμενο. Έτσι είναι πολύ εύκολο να δημιουργήσει κανείς με το RMI πολύ ευέλικτες αρχιτεκτονικές όπου όλες οι απαραίτητες κλάσεις τοποθετούνται σε ένα γνωστό URL και εγκαθίστανται στους διάφορους host δυναμικά.

2.3. CORBA

2.3.1 Γενικά

Το **CORBA (Common Object Request Object Architecture)** παρέχει και αυτό ένα πλαίσιο για την ανάπτυξη και εκτέλεση κατανεμημένων εφαρμογών αντικειμένων. Επιπλέον ορίζει ένα σύνολο κανόνων με το οποίο τα διάφορα αντικείμενα που συνεργάζονται σε μια κατανεμημένη εφαρμογή μπορεί να έχουν γραφεί σε διαφορετικές γλώσσες προγραμματισμού και να 'τρέχουν' σε διαφορετικές πλατφόρμες. Παρέχει ακόμα ένα ευρύ φάσμα βοηθητικών υπηρεσιών που ως στόχο έχουν την ευκολότερη ανάπτυξη και λειτουργία των εφαρμογών αυτών. Οι υπηρεσίες αυτές είτε είναι γενικές, συμπεριλαμβάνονται δηλαδή σε κάθε υλοποίηση του CORBA, είτε προσανατολίζονται σε ένα συγκεκριμένο πεδίο εφαρμογής. Το βασικό στοιχείο που πρέπει να γίνει εδώ κατανοητό είναι ότι το CORBA δεν είναι κάτι το υπαρκτό με την έννοια ενός προγράμματος. Είναι ένα σύνολο προδιαγραφών το οποίο αναλαμβάνουν να υλοποιήσουν διάφορες εταιρείες λογισμικού.

Το CORBA έχει προέρθει από την τάση για μετακίνηση προς ανοικτά υπολογιστικά περιβάλλοντα, δηλαδή προς περιβάλλοντα που δεν εξαρτώνται από συγκεκριμένες αρχιτεκτονικές υλικού, λειτουργικά συστήματα, γλώσσες προγραμματισμού. Σε ένα 'ανοικτό υπολογιστικό περιβάλλον', οι διάφοροι παροχείς λογισμικού συμφωνούν να αναπτύξουν τα προϊόντα τους σύμφωνα με κάποιο καθιερωμένο πρότυπο το οποίο αναπτύσσεται και διατηρείται συνήθως από έναν οργανισμό, που πολύ συχνά αναλαμβάνει και ρόλο μεσολαβητή. Η τάση αυτή έχει βοηθηθεί, αν όχι προέλθει από την ανάπτυξη του Internet και του World Wide Web. Σε ένα τέτοιο περιβάλλον υπάρχει απόλυτη ανάγκη για λογισμικό που

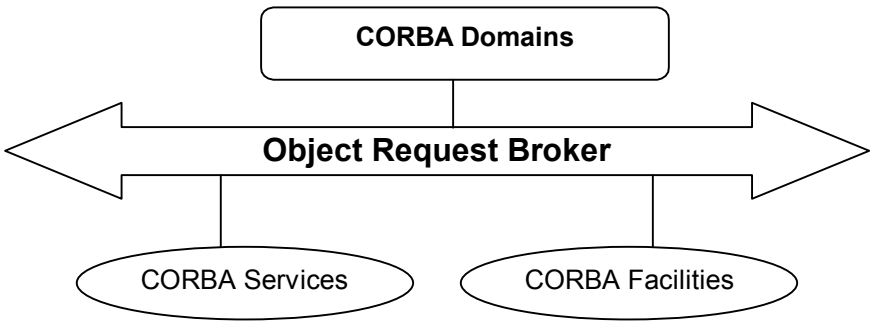
δεν βασίζεται σε μία μόνο τεχνολογία αλλά είναι όπως είπαμε ‘ανοικτό’. Επίσης σημαντικός παράγοντας για την μετακίνηση σε ανοικτά υπολογιστικά περιβάλλοντα είναι η προστασία της επένδυσης μιας εταιρείας σε μία τεχνολογία, η οποία πλέον έχει ξεπεραστεί. Ως ένα παράδειγμα σε αυτό μπορεί να αναφερθεί η ανάγκη για υποστήριξη πάρα πολλών συστημάτων που έχουν γραφεί σε COBOL, γλώσσα ευρύτατα διαδεδομένη πριν 2 περίπου δεκαετίες. Τα προγράμματα αυτά έχουν αποδειχθεί στην πράξη ότι λειτουργούν σωστά και έτσι γλιτώνεται και κόπος και χρήμα αν αυτά θα μπορούσαν να διατηρηθούν. Πρέπει όμως να μπορούν να χρησιμοποιούνται και για νέες χρήσεις που δεν είχαν προβλεφθεί όταν αυτά αναπτύσσονταν (πχ. διεπαφή με το Web). Το κενό αυτό είναι που προσπαθεί να καλύψει το CORBA.

Το CORBA λοιπόν, αναπτύσσεται, υποστηρίζεται και προωθείται από τον **OMG (Object Management Group)** που είναι ίσως ο μεγαλύτερος φορέας λογισμικού στον κόσμο. Ο κύριος σκοπός του OMG είναι η επίλυση των προβλημάτων που προκύπτουν κατά την ανάπτυξη και σχεδίαση μεγάλων συστημάτων λογισμικού. Αποτελείται συγκεκριμένα από 800 εταιρείες, τόσο από τον χώρο της πληροφορικής, όσο και από αλλα πεδία. Ιδρύθηκε το 1989 από τις εταιρείες 3COM, American Airlines, Canon Inc., Data General, Hewlett-Packard, Philips Telecommunications, Sun Microsystems, Unisys Corporation. Ο OMG δεν παράγει καθόλου λογισμικό με την μορφή εκτελέσιμων προγραμμάτων. Αντίθετα παράγει προδιαγραφές λογισμικού (specifications) οι οποίες γράφονται σε μια δηλωτική γλώσσα την **IDL** (*Interface Definition Language*). Αυτές οι προδιαγραφές καλύπτουν διάφορες ανάγκες που έχουν παρατηρηθεί από κάποιες εταιρείες – μέλη του. Γενικά πάντως παρόλο που ο OMG παράγει μόνο προδιαγραφές, μια πρόταση δεν μπορεί να γίνει επίσημη προδιαγραφή του OMG αν κάποιο μέλος του δεν έχει αναπτύξει μία πλήρως λειτουργική υλοποίηση της.

Το CORBA αποτελείται από 4 ειδών προδιαγραφές, οι οποίες αναφέρονται συνολικά ως **OMA (Object Management Architecture)**. Αυτές είναι:

1. ORB (Object Request Broker).
2. CORBAServices (COS).
3. CORBAFacilities.
4. CORBADomains.

Η διάταξη τους μπορεί να περιγραφεί ως εξής:



Σχήμα 2 Διάταξη των Προδιαγραφών του CORBA

Παρακάτω θα παρουσιάσουμε αναλυτικά κάθε ένα από τα συστατικά του.

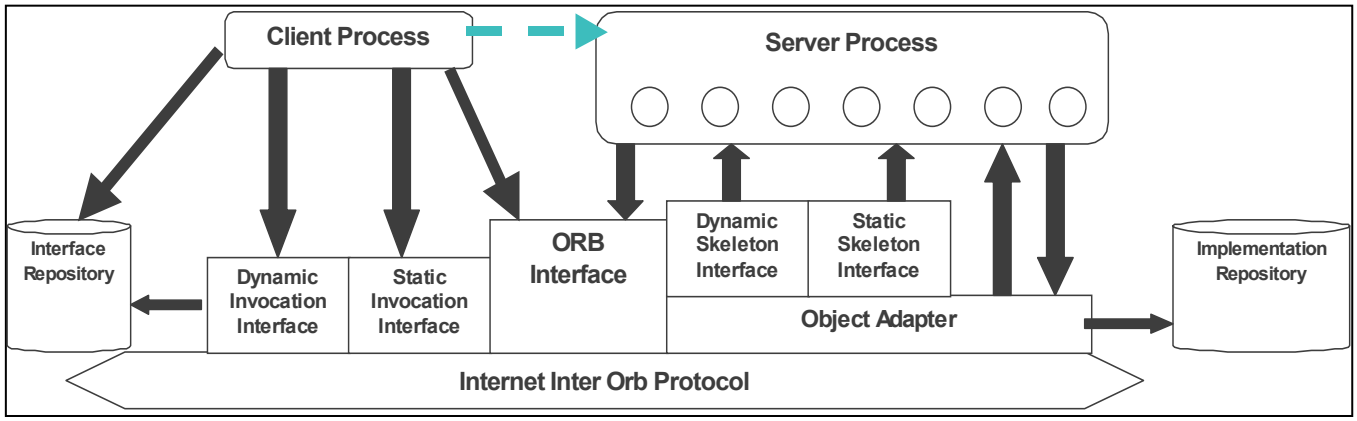
2.3.2 Object Request Broker.

Ο ORB είναι το πιο σημαντικό συστατικό της αρχιτεκτονικής του CORBA. Καθορίζει έναν δίαυλο επικοινωνίας μέσω του οποίου μπορούν να επικοινωνήσουν ετερογενή αντικείμενα ακολουθώντας το μοντέλο πελάτη - εξυπηρετητή. Κάθε αντικειμένο μπορεί να καλέσει ή να δεχθεί κλήσεις μεθόδων μέσω του ORB από αντικειμένα που βρίσκονται τοπικά ή σε διαφορετικές μηχανές. Κάθε αντικείμενο δεν ενδιαφέρεται για την τοποθεσία ενός άλλου αντικειμένου με το οποίο αλληλεπιδρά, ούτε για τον τρόπο υλοποίησης του. Μπορεί κάλλιστα ένα αντικείμενο που έχει υλοποιηθεί σε Java να καλέσει μεθόδους σε ένα αντικείμενο που έχει γραφτεί σε C++ ή ακόμα και COBOL. Το μόνο που χρειάζεται να γνωρίζει είναι ένα interface, το οποίο περιγράφει όλες τις υπηρεσίες που μπορεί να παρέχει ο εξυπηρετητής στον πελάτη, το οποίο έχει υλοποιηθεί στην περιγραφική γλώσσα IDL, όπου όπως προαναφέραμε συντάσσονται όλες οι προδιαγραφές του OMG. Το ORB, λοιπόν, είναι αυτό που θα δεχθεί την κλήση, θα εντοπίσει το ζητούμενο αντικείμενο, θα περάσει τις παραμέτρους μέσω δικτύου, θα καλέσει την μέθοδο και θα επιστρέψει τα αποτελέσματα στον πελάτη.

Οι πρώτες υλοποιήσιμες προδιαγραφές για ORBs από τον OMG εκδόθηκαν το 1992 με το CORBA 1.1 (η πρώτη λειτουργική περιγραφή του CORBA – οι προηγούμενες ήταν δοκιμαστικές). Οι προδιαγραφές αυτές δεν είχαν και πολύ μεγάλη επιτυχία καθώς περιείχαν ένα παράδοξο. Επέτρεπαν μεν σε ετερογενή αντικείμενα να επικοινωνούν μεταξύ τους αρκεί να γινόταν χρήση του ίδιου ORB. Δηλαδή δεν επέτρεπε σε εφαρμογές που χρησιμοποιούσαν διαφορετικά ORBs να συνεργαστούν. Το CORBA 2.0 προσέθεσε ένα πολύ σημαντικό στοιχείο το GIOP (General Inter Orb Protocol) που επέτρεπε την επικοινωνία διαφορετικών ORBs. Το IIOP αποτελεί μια υλοποίηση του GIOP σε δίκτυα TCP/IP. Το CORBA 2.0 είναι η έκδοση που κυριαρχεί σήμερα στην αγορά. Συγκεκριμένα τα περισσότερα προϊόντα υλοποιούν

την έκδοση 2.2 ενώ τον τελευταίο καιρό έχουν αρχίσει να κυκλοφορούν ORBs που είναι σύμφωνα με την έκδοση 2.3. Για τον σκοπό αυτό θα μελετήσουμε την αρχιτεκτονική ενός ORB όπως περιγράφεται από το CORBA 2 specification (σχήμα 3). Σε πρώτη ματιά, παρατηρούμε τον πλούσιο αριθμό συστατικών από τα οποία αποτελείται ένα CORBA ORB (σχήμα 3) σε σχέση με ένα RMI ORB – κατά την ορολογία του CORBA (σχήμα 1).

Σχήμα 3. Η δομή ενός CORBA 2.0 ORB



Όπως φαίνεται στο παραπάνω διάγραμμα ο ORB αποτελείται από αρκετά στοιχεία. Αυτά ναι μεν ορίζονται από τις προδιαγραφές του CORBA, αλλά δεν υπάρχουν όλα σε κάθε υλοποίηση που κυκλοφορεί στο εμπόριο. Η πρόσβαση στα συγκεκριμένα συστατικά στοιχεία από τα διάφορα προγράμματα γίνεται με κλήση των μεθόδων που ορίζονται από το συγκεκριμενο API (Application Programming Interface).

- **Static Invocation Interface (SII):** Δίνει την δυνατότητα στον client να καλέσει προκαθορισμένες μεθόδους ενός server object (servant). Λέγοντας προκαθορισμένες, εννοούμε ότι οι μέθοδοι αυτοί έχουν οριστεί πριν την μεταγλώττιση των προγραμμάτων. Ο ορισμός γίνεται με την χρήση IDL. Μετά εφαρμόζουμε στα αρχεία IDL ένα εργαλείο που έρχεται μαζί με το ORB και έχει το συμβατικό όνομα IDL compiler (διαδικασία τυπική για τα κατανεμημένα συστήματα αντικείμενων). Το εργαλείο αυτό είναι διαφορετικό για κάθε γλώσσα προγραμματισμού στην οποία θα αναπτυχθεί η υλοποίηση των interfaces αυτών. Το εργαλείο αυτό παράγει τα τους στατικούς αντιπροσώπους (static stubs και skeletons για τον client και server) αντίστοιχα. Η εφαρμογή καλεί την επιθυμητή μέθοδο πάνω στο stub (δηλ. έχουμε τοπική κλήση μεθόδου). Το stub εδώ, όπως

και στο RMI, υλοποιεί την μεταβίβαση και προώθηση της κλήσης στο απομακρυσμένο αντικείμενο (το γνωστό marshalling).

- **Dynamic Invocation Interface (DII):** Ορίζει ένα API το οποίο επιτρέπει την κλήση *μη προκαθορισμένων* μεθόδων, δηλαδή μεθόδων που δεν έχουν οριστεί πρίν από την μεταγλώττιση των προγραμμάτων. Ουσιαστικά αυτό το συστατικό επιτρέπει την δυναμική ανακάλυψη υπηρεσιών που έχουν οριστεί και πάλι με IDL, την δημιουργία των παραμέτρων που αυτές απαιτούν, την δημιουργία της κλήσης και την απόκτηση των αποτελεσμάτων. Οι δυναμικές κλήσεις είναι πιο δύσκολες στον προγραμματισμό, πιο αργές από τις στατικές, αλλά παρέχουν ευελιξία. Επίσης, δεν παρέχεται δυνατότητα χρήσης τους σε όλες τις υλοποιήσεις των ORBs. Σε κάθε περίπτωση πάντως, το αν μία κλήση θα είναι στατική ή δυναμική αφορά μόνο τον client. Η άλλη πλευρά δεν καταλαβαίνει την διαφορά.
- **Interface Repository:** Ουσιαστικά είναι μία βάση δεδομένων όπου τοποθετούνται IDL interfaces, σε binary μορφή, ώστε να μπορούν να ανακτηθούν δυναμικά από τα διάφορα προγράμματα. Περιέχει τα πάντα που περιγράφονται στο IDL για τις μεθόδους και τις παραμέτρους τους (*method signatures*). Με την χρήση του Interface Repository κάθε συστατικό, είτε αυτό προέρχεται από μια εφαρμογή, είτε είναι ενσωματωμένο στο CORBA, μπορεί να ανακτηθεί δυναμικά, κάτι που βοηθά σημαντικά στην ευελιξία του όλου συστήματος. Κάθε συστατικό που αποθηκεύεται στο Interface Repository αποκτά μία *Repository ID*, έτσι ώστε να μπορεί να αναγνωρίζεται μοναδικά και καθολικά για κάθε υλοποίηση ORB και Interface Repository. Οι συγκεκριμένοι προσδιοριστές παράγονται αυτόματα από το σύστημα και ακολουθούν συμβάσεις καθορισμένες από τον OMG, δίνοντας έτσι πλήρη συμβατότητα.
- **To ORB interface:** Δίνει την δυνατότητα σε κάποιον client ή server να επικοινωνεί απευθείας με το ORB για διάφορες λειτουργίες γενικής χρήσης (για παράδειγμα μετατροπή object reference σε string.)

Τα παραπάνω χρησιμοποιούνται κυρίως από τις εφαρμογές client. Στην άλλη πλευρά έχουμε αντίστοιχα συστατικά.

- **Static Skeleton Interface (SII):** Είναι το αντίστοιχο του SII στην πλευρά του server. Παράγεται από τον IDL compiler και δίνει την δυνατότητα σε ένα αντικείμενο να δεχθεί κλήσεις για προκαθορισμένες μεθόδους.
- **Dynamic Skeleton Interface (DSI):** Είναι το αντίστοιχο του DII στην πλευρά του server. Ένα από τα νέα χαρακτηριστικά του CORBA 2.0, δίνει την δυνατότητα στην server process να λαμβάνει εισερχόμενα μηνύματα (κλήσεις μεθόδων) για αντικείμενα που δεν έχουν μεταγλωττισμένα IDL interfaces, που

δεν έχουν δηλαδή skeletons. Το DSI κοιτά τις παραμέτρους σε εισερχόμενες κλήσεις και αποφασίζει σε ποιο από τα αντικείμενα που ελέγχει απευθύνεται και σε ποια μέθοδο του αντιστοιχεί.

- **Object Adapter:** Είναι ίσως το πιο σημαντικό συστατικό του ORB στην πλευρά του server. Όπως φαίνεται από το σχήμα 2 έρχεται σε άμεση επαφή με το καθαρά δικτυακό τμήμα του ORB. Άρα είναι το πρώτο που λαμβάνει μία κλήση μεθόδου. Έτσι είναι υπεύθυνο για την αρχικοποίηση του server object στο οποίο η κλήση μεθόδου απευθύνεται όταν αυτό δεν είναι ενεργό. Δίνει έτσι την ψευδαίσθηση στους διαφόρους clients ότι τα αντικείμενα που παρέχουν τις υπηρεσίες είναι πάντα ενεργά περιμένοντας για κλήσεις. Επίσης αναθέτει σε κάθε server object μία object reference (μοναδική ταυτότητα αντικειμένου) που διευκολύνει τον εντοπισμό του. Τέλος διαχειρίζεται το **Implementation Repository,** το οποίο θα περιγραφεί στην συνέχεια. Σύμφωνα με το CORBA 2.0 κάθε ORB πρέπει να υποστηρίζει ένα βασικό αριθμό χαρακτηριστικών. Το σύνολο αυτό των χαρακτηριστικών συνιστά τον **Basic Object Adapter (BOA).** Περισσότερα στοιχεία για την λειτουργία του BOA παρατίθενται στην συνέχεια.
- **Implementation Repository:** Είναι μία Βάση Δεδομενων η οποία περιέχει πληροφορίες σχετικά με τα διάφορα αντικείμενα που υποστηρίζει ένας server.

Όπως προαναφέρθηκε το CORBA 1.1 είχε ένα μεγάλο μειονέκτημα και γι' αυτό δεν έτυχε ευρείας αποδοχής. Η πραγματική μεταφερσιμότητα μπορούσε να υπάρξει μόνο μεταξύ των εφαρμογών που χρησιμοποιούσαν ίδια υλοποίηση του ORB. Διαφορετικές υλοποιήσεις ORB δεν μπορούσαν να συνεργαστούν και κατά συνέπεια ούτε οι εφαρμογές που αυτές υποστηρίζαν. Το CORBA 2.0 έφερε το GIOP (General Inter Orb Protocol), το οποίο επέτρεπε την συνεργασία μεταξύ διαφορετικών ORBs, καθορίζοντας μια σειρά από κοινές μορφές μηνυμάτων και αναπαραστάσεις δεδομένων. Συγκεκριμένα καθορίζονται μια σειρά από 7 μορφές μηνυμάτων που χρειάζονται για την άλληλεπίδραση διαφορετικών ORBs. Επίσης ορίζει μια σειρά από κοινές αναπαραστάσεις (Common Data Representations) όλων των τύπων δεδομένων που μπορούν να ορίστουν μέσω του IDL, έτσι ώστε να μπορούν να μεταφερθούν σε ετερογενή δίκτυα. Το IIOP είναι μια υλοποίηση του GIOP για δίκτυα TCP / IP. Για να είναι ένα ORB συμβατό με το CORBA 2.0 *(CORBA 2.0 compliant)* πρέπει να υποστηρίζει το IIOP με κάποιον τρόπο είτε ως το δικό του πρωτόκολλο ανταλλαγής μηνυμάτων, είτε παρέχοντας ένα επίπεδο μετάφρασης. Γενικά πάντως με το CORBA 2.0 το IIOP είναι ο καθολικός σκελετός μέσω του οποίου θα επικοινωνούν όλα τα ORBs.

Για την ολοκλήρωση της περιγραφής του ORB μένει μία εκκρεμότητα. Αυτή αφορά την διαδικασία ενεργοποίησης ενός αντικειμένου που περιέχει μία μέθοδο, όταν λαμβάνεται μία κλήση της. Η ενεργοποίηση των αντικειμένων πρέπει να αποκρύπτεται από τον client, στον

οποίο πρέπει να δίνεται η εντύπωση ότι τα αντικείμενα τα οποία προσφέρουν τις υπηρεσίες που αυτός θέλει είναι πάντοτε ενεργά και περιμένουν κλήσεις. Όπως προαναφέραμε κάτι τέτοιο είναι ευθύνη του Object Adapter.

Υπάρχουν πολλοί δυνατοί τρόποι ενεργοποίησης ενός αντικειμένου, με την εκκίνηση μιας νέας διεργασίας, με την δημιουργία ενός νέου νήματος σε μία ήδη υπάρχουσα διεργασία, ή με την χρήση υπάρχοντος νήματος ή διεργασίας. Κάθε server[2] μπορεί να υποστηρίζει πολλούς adapters για κάθε ένα από αυτά τα είδη αρχικοποίησης. Ο Basic Object Adapter έχει ως σκοπό την ενσωμάτωση και υποστήριξη όλων των μηχανισμών ενεργοποίησης ενός server objects μέσα στις server processes. Συγκεκριμένα το CORBA 2.0 ορίζει 4 μηχανισμούς ενεργοποίησης:

1. **Shared Server.**Τα server objects βρίσκονται όλα σε μία διεργασία η οποία ενεργοποιείται μόλις ληφθεί η πρώτη κλήση μεθόδου για κάποιο από αυτά. Όταν γίνει αυτό χειρίζεται πλέον όλες τις κλήσεις. Όταν τερματίσει η συγκεκριμένη διεργασία τερματίζουν την λειτουργία τους όλα τα αντικείμενα που περιέχονται σε αυτήν.
2. **Unshared Server:** Κάθε server object εκτελείται σε μία δική του διεργασία. Πάλι η ενεργοποίηση γίνεται όταν ληφθεί η πρώτη κλήση για μια μέθοδο. Κάθε αίτηση για ένα αντικείμενο έχει ως αποτέλεσμα την ενεργοποίηση μιας νέας server process ακόμα και αν υπάρχει ήδη ενεργό ένα άλλο αντικείμενο.
3. **Server Per Method :** Κάθε κλήση μεθόδου οδηγεί σε ενεργοποίηση αντικειμένου με αποτέλεσμα πολλές υλοποιήσεις ίδιων αντικειμένων να συμβαίνει να είναι ταυτόχρονα ενεργές.
4. **BOA Persistent Server :** Κάθε server process δεν ενεργοποιείται από τον BOA αλλά με κάποιον άλλο τρόπο (πχ. από έναν χρήστη). Η εφαρμογή ενεργοποιείται με αυτόν τον τρόπο και όλες οι υπόλοιπες κλήσεις αντιμετωπίζονται όπως στον μηχανισμό 1.

Σε πρακτικό επίπεδο αξίζει να παρατηρήσουμε ότι ο ORB δεν είναι μια ξεχωριστή διεργασία. Αντίθετα είναι ενα σύνολο βιβλιοθηκών (πακέτων κλάσεων) που χρησιμοποιούνται από τα διάφορα αντικείμενα. Έτσι ο ORB 'ενσωματώνεται' κατά κάποιον τρόπο μέσα στο αντικείμενα που το χρησιμοποιούν . Δεν υπάρχει έτσι ανάγκη για επιπλέον κλήσεις για επικοινωνία με το ORB, κάτι που θα οδηγούσε σε καθυστερήσεις.

[2] Με τον όρο server process εννοούμε ένα πρόγραμμα . Με τον όρο server object εννοούμε ένα αντικείμενο το οποίο υλοποιεί ένα interface και παρέχει μία υπηρεσία.

2.3.3 CORBAServices (COS) Specification

Είναι μια προδιαγραφή που περιγράφει μια επιπλέον λειτουργία η οποία έρχεται να συμπληρώσει την λειτουργία του ORB, συνήθως στο 'επίπεδο αντικειμένου', σε χαμηλό επίπεδο δηλαδή. Είναι ένα σύνολο από υπηρεσίες και λειτουργίες οι οποίες έχουν ως σκοπό να **επεκτείνουν την λειτουργία του ORB**. Παρέχουν έτσι στους προγραμματιστές ένα σύνολο από μεθόδους οι οποίες μπορούν να κληθούν για εντοπισμό αντικειμένων, αποθήκευση αντικειμένων κτλ. Αναφέρονται κυρίως στην πλευρά του server χρησιμοποιούνται όμως επιτυχώς από όλα τα μέρη του κατανεμημένου συστήματος. Η περιγραφή τους γίνεται και πάλι χρησιμοποιώντας *IDL*. Οι CORBA Services που έχουν οριστεί ως τώρα περιλαμβάνουν :

Πίνακας 1. CORBA Services

Όνομα	Λειτουργία
1. Life Cycle Service	Παρέχει ένα API για την δημιουργία, καταστροφή, μεταφορά, διαγραφή αντικειμένων.
2.Persistence Service	Παρέχει ένα interface το οποίο δίνει την δυνατότητα αποθήκευσης αντικειμένων σε OODBMSs, RDBMSs ή σε οποιοδήποτε άλλο μέσο.
3.Naming Service	Παρέχει πρόσβαση σε ένα ιεραρχικό σύστημα ονοματολογίας αντικειμένων. Επιτρέπει την χρήση μοναδικών ονομάτων για τον εντοπισμό αντικειμένων. Η συγκεκριμένη υπηρεσία μπορεί να εντοπίσει αντικείμενα με βάση το όνομα τους. Χρησιμοποιείται κυρίως για την απόκτηση της πρώτης reference σε ένα αντικείμενο.
4.Event Service	Ορίζει το κανάλι γεγονότων (event channel) το οποίο συλλέγει και κατανέμει events μεταξύ αντικειμένων που έχουν δηλώσει ενδιαφέρον για αυτά. Τα διάφορα γεγονότα μεταφέρονται από πηγές γεγονότων σε ακροατές γεγονότων.
5.Concurrency Control Service	Παρέχει δυνατότητα συγχρονισμού της ταυτόχρονης πρόσβασης πολλών clients σε διαμοιραζόμενους πόρους.
6.Transaction Service	API για χειρισμό transactions.
7.Relationship Service	Παρέχει ένα μηχανισμό για δημιουργία δυναμικών συσχετισμών μεταξύ αντικειμένων.

8.Externalization Service	Ορίζει έναν standard τρόπο για την επικοινωνία δεδομένων από ένα component Επιτρέπει σε ένα αντικείμενο ή ένα σύνολο αντικειμένων να μετατραπεί σε ένα stream από bytes.(Στην Java αυτό γίνεται με το Object Serialization API).
9.Query Service	Όρίζει μια επέκταση της SQL (βασίζεται και στην OQL) για ερωτήσεις (queries) σε συλλογές αντικειμένων (object collections).
10.Licensing Service	Παρεχει ένα μοντέλο για έλεγχο χρήσης ενός component το οποίο επιβάλλει την διαφύλαξη της πνευματικής ιδιοκτησίας στα software components.
11.Properties Service	Δίνει την δυνατότητα ανάθεσης ιδιοτήτων σε ένα component. Συσχετίζει δηλαδή ζεύγη ονομάτων – τιμών με κάποιο αντικείμενο.
12.Time Service	Ορίζει μεθόδους για συγχρονισμό σε κατανεμημένα περιβάλλοντα. Με βάση αυτήν την υπηρεσία μπορούμε να έχουμε την τρέχουσα ώρα για κάποιο σύστημα καθώς επίσης και το περιθώριο λάθους. Δεν είναι δυνατόν να έχουμε την ακριβή ώρα λόγω πολλών παραγόντων όπως επίσης και της καθυστέρησης μεταβίβασης του μηνύματος.
13.Security Service	Παρέχει ένα πλήρες σύνολο για ορισμό υπηρεσιων ασφάλειας.
14.Trader Service	Επιτρέπει στα αντικείμενα να ‘δημοσιοποιούν’ τις μεθόδους τους τους ώστε να είναι δυνατή η χρήση του DII. Με απλά λόγια, επιτρέπει τον εντοπισμό υπηρεσιών (object servers) με βάση την λειτουργικότητα και όχι μόνο το όνομα όπως η Naming Service.
15.Collection Service	CORBA interfaces για ένα σύνολο από δομές δεδομένων.

Στο πεδίο των διαφόρων υπηρεσιών υπάρχει μια συνεχής δραστηριότητα από τον OMG καθώς γίνεται συνεχής προσπάθεια για τον ορισμό νέων υπηρεσιών. Οι πιο πρόσφατες αφορούν Firewalls και Fault Tolerance services.

2.3.4.CORBAFacilities Specification

Και εδώ καθορίζεται μία επιπλέον λειτουργικότητα για το περιβάλλον του **CORBA** σε ‘υψηλότερο’ όμως επίπεδο από αυτήν μιας CORBAService, πιο κοντά δηλαδή στον χρήστη. Ο σκοπός άλλωστε των συγκεκριμένων υπηρεσιών είναι ο προσανατολισμός του CORBA στις ανάγκες συγκεκριμένων εφαρμογών. Έχουν οριστεί και πάλι με την χρήση IDL. Διακρίνονται σε ‘Οριζόντιες’ και ‘Κάθετες’ υπηρεσίες. Συγκεκριμένα μία ‘κάθετη’ υπηρεσία

έχει συγκεκριμένες εφαρμογές σε ένα πεδίο. Εώς σήμερα έχουν οριστεί οι εξής 8 ‘κάθετες’ υπηρεσίες :

- **Accounting :** Για εμπορικές συναλλαγές αντικειμένων.
- **Application Development :** Επικοινωνία μεταξύ αντικειμένων που χρησιμοποιούνται στη ανάπτυξη εφαρμογών.
- **Distributed Simulation:** Επικοινωνία για Αντικείμενα που λειτουργούν σε κατανεμημένες προσομοιώσεις.
- **Imagery:** Επικοινωνία και συνεργασία συσκευών γραφικών και δεδομένων που αποτελούν εικόνες.
- **Information Superhighways:** Επιτρέπει την επικοινωνία μεταξύ εφαρμογών πολλών χρηστών σε Δίκτυα Ευρείας Περιοχής.
- **Manufacturing:** Συνεργασία μεταξύ αντικειμένων που χρησιμοποιούνται σε περιβάλλοντα κατασκευής.
- **Mapping:** Αντικείμενα που χρησιμοποιούνται για χαρτογραφήσεις.
- **Oil and Gas Industry Exploitation and Production:** Επικοινωνία μεταξύ αντικειμένων που χρησιμοποιούνται στην αγορά πετρελαίου.

Οι ‘οριζόντιες’ υπηρεσίες μπορούν να χρησιμοποιηθούν από τις περισσότερες εφαρμογές. Ως τώρα υπάρχουν 4 κατηγορίες.

- **User Interface**: Προφανώς αναφέρεται στο σύστημα διεπαφής μιας εφαρμογής με τον χρήστη.
- **Information Management:** Αφορούν την μοντελοποίηση, τον ορισμό, την αποθήκευση την ανταλλαγή και την ανάκτηση πληροφοριών.
- **System Management:** Διαχείριση Πληροφοριακών Συστημάτων.
- **Task Management:** Αυτοματοποίηση διαφόρων εργασιών χρηστών ή συστήματος.

2.3.5 CORBADomain Specification

Περιγράφει με λεπτομέρεια κάποιο είδος επιπλέον λειτουργικότητας για το CORBA σε ό,τι αφορά ένα συγκεκριμένο πεδίο εφαρμογής (πχ. Πληροφορική Υγείας). Απευθύνονται δηλαδή σε έναν συγκεκριμένο τομέα για τον οποίο μπορούν να μοντελοποιούν και ολόκληρες διαδικασίες του. Επιβλέπονται από τις αντίστοιχες ομάδες εργασίες του **OMG.** Αποτελούν ίσως το πιο σημαντικό στοιχείο για την εξέλιξη του CORBA και του OMG. Απώτερος στόχος δεν είναι να μάθουν όλοι οι προγραμματιστές CORBA [4]. Ο στόχος είναι να ενσωματωθεί το CORBA σε διεργασίες συγκεκριμένων πεδίων εφαρμογής με τέτοιον τρόπο έτσι ώστε τα τεχνικά στοιχεία που περιγράφονται σε αυτήν την ενότητα να μην απασχολούν πολύ κόσμο.

Αυτό υλοποιείται από τον OMG στα πλαίσια του **Business Object Framework (BOF)** από την **Business Object Task Force.** Ένα *business object* είναι ένα αντικείμενο επιπέδου εφαρμογής που μπορεί να χρησιμοποιηθεί με απρόβλεπτους συνδυασμούς [4]. Με απλά λόγια είναι ένας τρόπος για περιγραφή εννοιών που αφορούν τις εφαρμογές με τρόπο ανεξάρτητο από μία συγκεκριμένη εφαρμογή. Στόχος, για παράδειγμα, είναι ο ορισμός του business object *'πελάτης'* άσχετα με το αν έχουμε πελάτη ξενοδοχείου,ιατρείου και ανεξάρτητα από την εφαρμογή που θα κάνει χρήση της έννοιας του πελάτη. Το BOF έχει ως σκοπό των σχεδιασμό συστημάτων που μιμούνται τις πραγματικές δραστηριότητες που υποστηρίζουν. Το απώτερο αποτέλεσμα όλης αυτής της προσπάθειας θα είναι η παραγωγή κώδικα που θα είναι **πραγματικά** επαναχρησιμοποίησιμος.

2.3.6 CORBA 3.0

Το CORBA 3.0 είναι η νέα έκδοση των προδιαγραφών του CORBA. Μερικά από τα χαρακτηριστικά του έχουν ήδη ενσωματωθεί σε αναβαθμίσεις του CORBA (2.1, 2.2, 2.3). Το CORBA 3.0 θα υποστηρίζει πέρασμα παραμέτρων αντικειμένων by value. Επίσης θα παρέχει δυνατότητα για δημιουργία κώδικα IDL από Java interfaces (Java - IDL mapping) εκτός από το αντίστροφο, που γίνεται σήμερα. Θα παρέχει συμβατότητα με το DCOM, το αντίστοιχο κατανεμημένο σύστημα αντικειμένων το οποίο προωθεί η Microsoft. Ακόμα θα παρέχει δυνατότητα για χρήση URLs για προσδιορισμό αντικειμένων εκτός από τα IORs που υπάρχουν στο CORBA 2.0 (Interoperable Naming Service). Η μορφή αυτών των URLs θα είναι ***iioploc://iiopname,*** όπου *iioploc* θα καθορίζει το πρωτόκολλο του URL και *iiopname* την συγκεκριμένη Naming Service η οποία θα παρέχει την object reference.

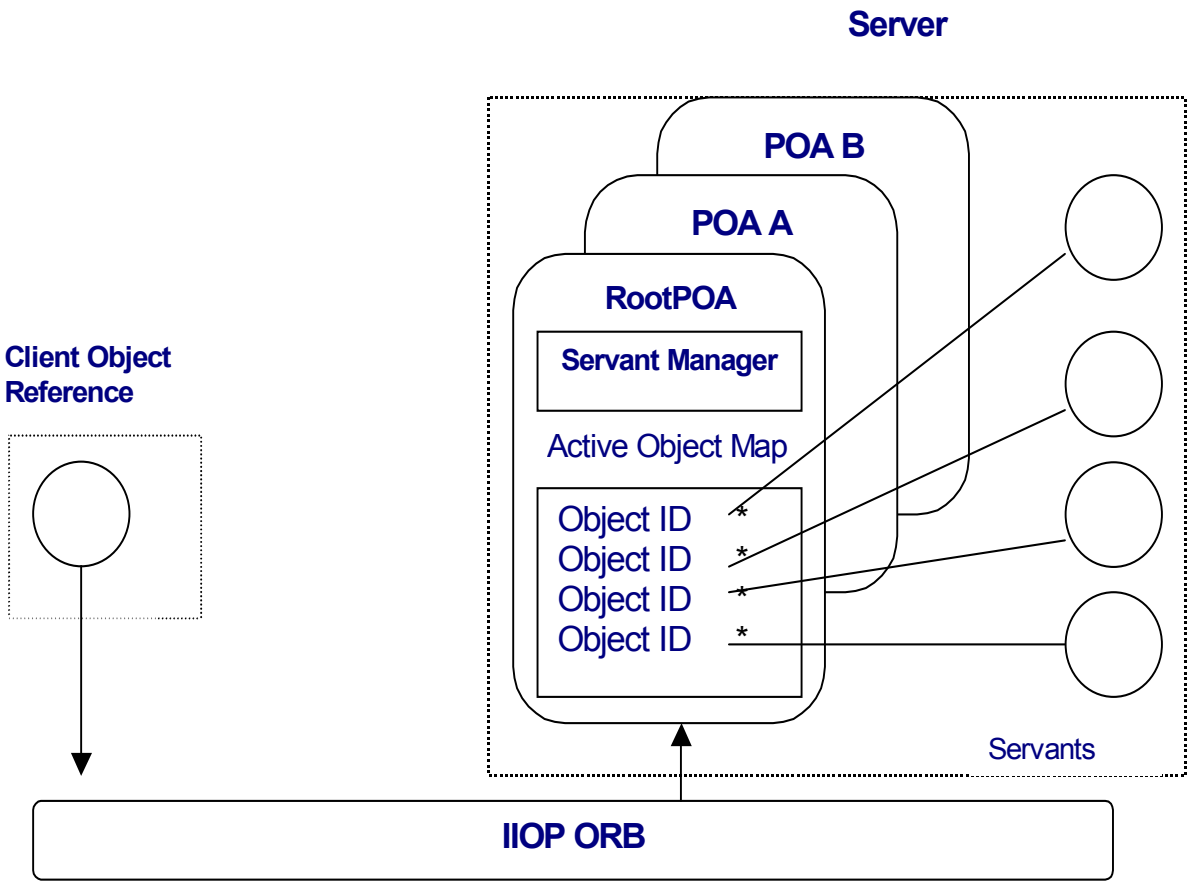
Ακόμα, παρέχει μια νέα έκδοση του BOA, η οποία ονομάζεται **POA** (Portable Object Adapter) και έχει στόχο την συμβατότητα μεταξύ των server εφαρμογών. Λέγοντας μεταφερσιμότητα εδώ εννούμε την δυνατότητα των εφαρμογών να λειτουργήσουν χωρίς αλλαγές σε διαφορετικά ORBs. Το CORBA 2.0 κατάφερε να επιτύχει συμβατότητα μεταξύ ORBs, και στην πλευρά του πελάτη. Στην αντίπερα όχθη όμως οι server applicationς δεν διακρίνονται από συμβατότητα. Αυτό συμβαίνει καθώς οι διάφορες εταιρείες έκριναν την έκδοση του BOA που πρότεινε ο OMG ελλιπή για πραγματική χρήση. Έτσι την εμπλούτισαν με δικά τους στοιχεία με αποτέλεσμα την ασυμβατότητα των εφαρμογών. Οι επεκτάσεις αυτές ήταν τόσο ασύμβατες που ο POA δεν αποτελεί βελτίωση του BOA, αλλά έχει αναπτυχθεί από την αρχή. Καθώς ο POA είναι παρών στις νέες εκδόσεις των πιο χρησιμοποιούμενων ORBs θα γίνει περιγραφή σε περισσότερη έκταση. Με την εισαγωγή του POA δεν θα εξαφανιστεί ο BOA. Αντίθετα μάλιστα. Ο BOA θα υπάρχει για κάποιο διάστημα παράλληλα με τον POA, ώστε να υποστηριχτούν όλες οι εφαρμογές που τον

χρησιμοποποιούν. Παρ'όλα αυτά, ο OMG στις προδιαγραφές του προτείνει όλες οι νέες εφαρμογές να χρησιμοποιούν τον POA.

Προχωρώντας στην περιγραφή του POA, ξεκινάμε λέγοντας ότι βασικά υποστηρίζει δύο κατηγορίες αντικείμενων : *transient* και *persistent*. Τα πρώτα τερματίζουν την λειτουργία τους με την server process που τα δημιούργησε χάνοντας έτσι την κατάσταση τους. Αντίθετα τα άλλα διατηρούν την κατάσταση τους, έχοντας την δυνατότητα να γράφουν τις τιμές των ιδιοτήτων τους πριν τερματίσει η διεργασία που τα δημιούργησε και όταν ξαναενεργοποιηθούν να τις ανακτούν. Επίσης ο POA εισάγει μία νέα έννοια: αυτή του *servant manager*. Στην ορολογία του CORBA, servant είναι ένα αντικείμενο που υλοποιεί ένα interface, που παρέχει δηλαδή την υλοποίηση μιας υπηρεσίας σε κάποιον client. Αυτός ο όρος χρησιμοποιείται για να τονισθεί η αντίθεση με τον όρο server, ο οποίος αναφέρεται τυπικά στο πρόγραμμα που δημιουργεί τους servants. Ο σκοπός λοιπόν των servant managers στον POA είναι να διαχειρίζονται την δημιουργία, ενεργοποίηση και απενεργοποίηση νέων servants. Ο σκοπός της εισαγωγής της έννοιας του servant manager, είναι να επιτρέψει στον προγραμματιστή να ελέγξει λεπτομέρειες που παλαιότερα αποκρύπτονταν από τον BOA. Μπορούν έτσι να δημιουργηθούν διάφορες υλοποιήσεις servant managers, εκτός από τις προκαθορισμένες που θα παρέχει ο κάθε POA, για να καλύψουν συγκεκριμένες ανάγκες. Η διαχείριση λοιπόν των servants από τους servant managers γίνεται με την βοήθεια ενός πίνακα που αντιστοιχίζει ενεργούς servants σε object Ids. Ο πίνακας αυτός ονομάζεται Active Object Map. Οι ταυτότητες των αντικειμένων μπορούν να δημιουργηθούν είτε από τον POA εσωτερικά, ή μπορούν να ανατεθούν άμεσα από την εφαρμογή.

Ένα από τα πιο σημαντικά σημεία βελτίωσης σε σχέση με τον BOA, είναι ότι δίνεται έλεγχος αρκετών λειτουργιών στον προγραμματιστή, ο οποίος καλείται να τις χρησιμοποιήσει όπως θέλει. Αυτό γίνεται μέσω της έννοιας της *'Policy'* που μπορεί να οριστεί ως ένα χαρακτηριστικό του POA το οποίο μπορεί να τύχει τροποποίησης από μια εφαρμογή. Οι διάφορες 'πολιτικές' λοιπόν ασχολούνται το ποιος αναθέτει ταυτότητες στα αντικείμενα (δηλ. ο POA ή η εφαρμογή), με το αν ένα αντικείμενο μπορεί να έχει μία ή πολλαπλές ταυτότητες, αν ένα αντικείμενο είναι transient ή persistent, με τον τρόπο ενεργοποίησης των αντικειμένων, με τον χειρισμό των servant managers κτλ. Όλες αυτές οι λεπτομέρειες τώρα μπορούν να ρυθμιστούν από τον προγραμματιστή.

Η δομή του POA μπορεί να αναπαρασταθεί από το παρακάτω σχήμα:



Σχήμα 4. Η δομή ενός POA.

Ο Root POA που φαίνεται στο σχήμα είναι ο αρχικός POA, ο οποίος λαμβάνεται από τον ORB και έχει προκαθορισμένες πολιτικές. Από αυτόν μπορούν να δημιουργηθούν επιπλέον POAs κάθε ένας από τους οποίους θα έχει ένα διαφορετικό σύνολο τιμών για τις πολιτικές που προαναφέραμε.

Άλλα στοιχεία του CORBA 3.0 αφορούν την υποστήριξη firewalls από το IIOP. Συγκεκριμένα ορίζει ως well – known ports το 683 για το IIOP και το 684 για το IIOP – SSL με αποτέλεσμα να υπάρχει δυνατότητα για υποστήριξη του IIOP από firewalls.
Σημαντικές προσθήκες θα γίνουν επίσης και στο επίπεδο των CORBA Services και CORBA facilities.

2.3.7 Interface Definition Language

Το IDL είναι η Esperanto του λογισμικού [16]. Δ**εν** μπορεί να θεωρηθεί ως γλώσσα προγραμματισμού, αλλά είναι μία γλώσσα για **περιγραφή** της δομής και της λειτουργίας ενός αντικειμένου. Δεν παρέχει δυνατότητες (όπως για παράδειγμα, δομές ελέγχου ροής, δομές επανάληψης) οι οποίες καθιστούν δυνατή την **υλοποίηση** ενός προγράμματος. Αρκείται μόνο

στην περιγραφή του interface ενός αντικειμένου, δηλαδή δηλώνει ποιές είναι οι ιδιότητες και ποιές οι λειτουργίες του.

Η συγγραφή του κώδικα IDL για κάθε αντικείμενο είναι ίσως το πρώτο βήμα για την ανάπτυξη ενός συστήματος CORBA, μετά την όλη σχεδίαση βέβαια. Ακολουθεί η χρήση ενός εργαλείου το οποίο διαθέτουν όλες οι υλοποιήσεις του CORBA, του **IDL compiler**. Ο IDL compiler επιλέγεται ανάλογα με την γλώσσα που έχει επιλεχθεί για την συγγραφή του κώδικα του αντικειμένου. Έτσι υπάρχουν τέτοια εργαλεία για πολλές γλώσσες προγραμματισμού όπως για την Java και την C++ ακόμα και για μη – αντικειμενοστρεφείς γλώσσες προγραμματισμού, όπως η C και η COBOL. Το συγκεκριμένο πρόγραμμα λοιπόν παίρνει ως είσοδο ενα αρχείο το οποίο περιέχει την περιγραφή σε IDL του interface ενός αντικειμένου και κάνει την μετατροπή στην γλώσσα υλοποίησης. Η συγκεκριμένη μετατροπή γίνεται σύμφωνα με τις αντιστοιχίσεις IDL (*IDL mappings*) που έχει ορίσει ο OMG για τις παραπάνω και όχι μόνο γλώσσες. Η έξοδος του IDL compiler για ένα συγκεκριμένο αρχείο IDL είναι κάποια αρχεία στην γλώσσα υλοποίησης. Τα πιο σημαντικά είναι τα αρχεία stub και skeleton. Το stub θα χρησιμοποιηθεί από το πρόγραμμα client και το skeleton από το πρόγραμμα server. Τα αρχεία αυτά περιέχουν κώδικα στην γλώσσα υλοποίησης (Java, C++) ο οποίος είναι ουσιαστικά το interface που ορίστηκε μέσω του IDL.

Όπως προαναφέρθηκε υπάρχουν αντιστοιχίσεις του IDL σε πολλές γλώσσες προγραμματισμού. Μιας και στην συγκεκριμένη εργασία η γλώσσα υλοποίησης θα είναι η Java θα ασχοληθούμε με την αντιστοίχιση IDL σε Java (*IDL – to – Java mapping*). Τα πιο σημαντικά στοιχεία φαινονται στον παρακάτω πίνακα:

IDL	Java
char	char
octet	byte
boolean	boolean
string	java.lang.String
short	short
long	long
float	float
double	double
any	Corba.any
module	interface
attribute	private ιδιότητες και setter – getter methods
operation parameter	in- /out /inout
struct	κλάση με όλες τις ιδιότητες public
sequence	array
array	array

Πίνακας 2. IDL to Java Mapping

Από τον παρακάνω πίνακα μπορούμε να παρατηρήσουμε ότι ναι μεν για τους primitive τύπους δεδομένων η αντιστοίχιση είναι εύλογη. Όσο όμως οι τύποι δεδομένων γίνονται πιο πολύπλοκοι, τόσο δυσκολεύει και η περιγραφή τους με το IDL.

Σε γενικές γραμμές, μπορεί να ειπωθεί ότι το IDL (ή το ίδιο ή ίσως η ιδέα ότι κάποιος πρέπει να μάθει μια νέα γλώσσα) 'απωθεί' αρχικά αυτόν που θέλει να ασχοληθεί με το CORBA. Για τον σκοπό αυτό υπήρξαν προσπάθειες από τις διάφορες εταιρείες λογισμικού ώστε να *'παρακαμφθεί'* το στάδιο του IDL (κυρίως από την 'κοινότητα' της Java). Η πρώτη από αυτές έγινε από τις εταιρείες Netscape και Inprise (πρώην Borland) και είχε ως αποτέλεσμα το σύστημα Caffeine που επέτρεπε την μετατροπή Java σε IDL. Έτσι αν όλα τα αντικείμενα της εφαρμογής είχαν ως γλώσσα υλοποίησης την Java τότε δεν θα χρειαζόταν καθόλου το IDL.

2.3.8 RMI over IIOP

Το RMI πάνω από το IIOP συνδυάζει τις τεχνολογίες του RMI και του CORBA. Συγκεκριμένα επιτρέπει την ανάπτυξη κατανεμημένων συστημάτων αντικειμένων χρησιμοποιώντας μόνο την Java, τόσο για την περιγραφή του interface των διαφόρων αντικειμένων, όσο και για την υλοποίηση του κώδικα. Επίσης, όπως το RMI, επιτρέπει την

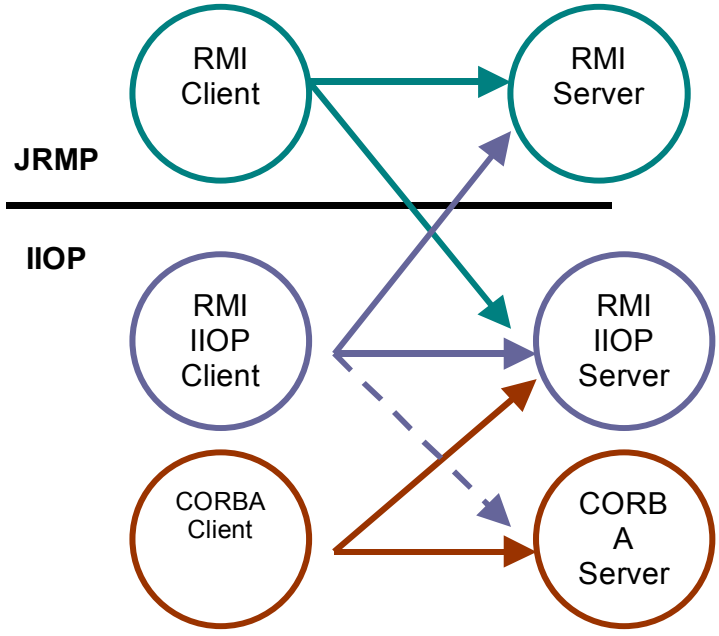
χρήση οποιουδήποτε αντικειμένου της Java ως παράμετρο σε μία remote μέθοδο (χρησιμοποιώντας *Java Serialization*). Έτσι μπορούν να περνούν παράμετροι και *by value* (RMI) και *by reference* (CORBA). Χρησιμοποιεί όμως το πρωτόκολλο του CORBA, το IIOP, αντί για το JRMP (Java Remote Method Protocol) του RMI, το οποίο επιτρέπει την συνεργασία μεταξύ αντικειμένων, τα οποία έχουν γραφεί σε διάφορες γλώσσες προγραμματισμού και τρέχουν σε διάφορα λειτουργικά συστήματα.

Η ενσωμάτωση των δύο παραπάνω τεχνολογιών είχε ως σκοπό να διατηρήσει την ευκολία προγραμματισμού αλλά και την πλούσια σημασιολογία του RMI, αλλά και να επεκτείνει το CORBA, ώστε να υποστηρίζει τα επιπλέον στοιχεία του RMI. (πχ. πέρασμα παραμέτρων αντικειμένων by - value). Για τον σκοπό αυτό ο OMG δέχτηκε να εισάγει στο CORBA με την έκδοση 2.3 τις παρακάτω νέες προδιαγραφές:

⇒ *Java – to – IDL mapping*: Επιτρέπει την μετατροπή των Java Interfaces σε IDL interfaces. Είναι σχεδόν η αντίστροφη διαδικασία από την IDL – to – Java mapping που περιγράψαμε νωρίτερα.

⇒ *Objects by Value:* Ορίζει ένα πρωτόκολλο ανάλογο με το Java Serialization, με το οποίο ένα αντικείμενο μπορεί να μετατρέψει την κατάσταση του σε ένα stream απο bytes, ικανών να μεταδοθούν σε ένα δίκτυο, επιτρέπωντας έτσι την χρήση του ως παράμετρος μίας μεθόδου.

Εκτός από την δυνατότητα για σύνδεση με το IIOP, παρέχει προς τα 'πίσω συμβατότητα' με το JRMP , δίνοντας έτσι δυνατότητα για διατήρηση των ήδη υπάρχοντων συστημάτων RMI. Συγκεκριμένα , είναι δυνατές οι παρακάτω αλληλεπιδράσεις [15] :



Σχήμα 5. Αλληλεπιδράσεις RMI – IIOP

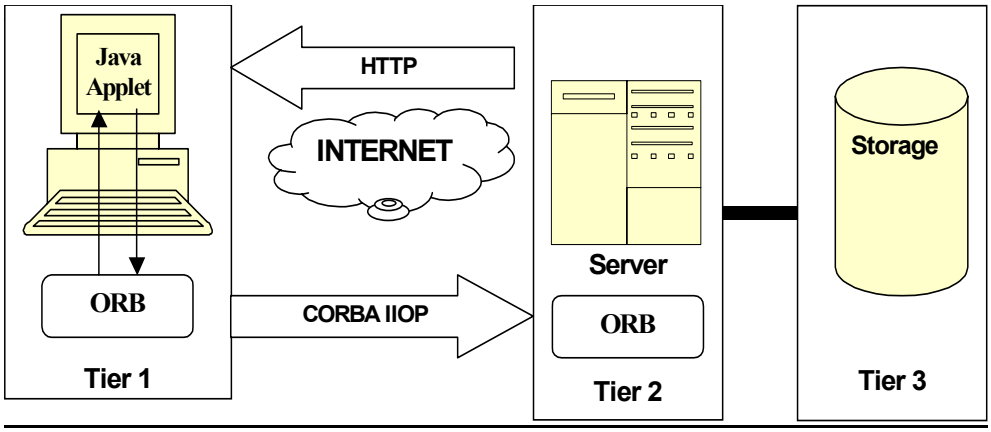
Στο συγκεκριμένο σχήμα παρατηρούμε ότι ένα client αντικείμενο που έχει γραφεί σε ‘καθαρό’ RMI (JRMP) μπορεί να έχει πρόσβαση σε ένα αντικείμενο που έχει γραφεί για RMI-IIOP. Εννοείται ότι η συμβατότητα αναφέρεται στις μεταγλωττισμένες εκδόσεις των παραπάνω αντικειμένων. Δεν χρειάζεται λοιπόν επαναμεταγλώττιση του κώδικα, μόνο η αλλαγή των τιμών σε κάποιες ιδιότητες της JVM. Ανάλογα και ένα αντικείμενο σε RMI - IIOP, μπορεί να έχει πρόσβαση σε ένα RMI – JRMP αντικείμενο. Τα πράγματα αλλάζουν κάπως όταν πρόκειται για αντικείμενα CORBA. Το βέλος με την διακεκομμένη γραμμή σημαίνει ότι ένα αντικείμενο RMI - IIOP δεν μπορεί να έχει πρόσβαση σε όλα τα **ήδη υπάρχοντα** αντικείμενα CORBA. Αυτό συμβαίνει καθώς το Java – to – IDL mapping δεν καλύπτει όλες τις δομές που μπορούν να οριστούν στο IDL. Στην περίπτωση ενός νέου αντικειμένου μπορούν να δημιουργηθούν από το εργαλείο compiler που διατίθεται και εδώ, τα αντίστοιχα interfaces σε IDL τα οποία μπορούν να υλοποιηθούν ως κανονικά αντικείμενα CORBA, πχ. σε C++.

2.4 Το OBJECT WEB.

Το **Object Web** είναι ένας όρος που χρησιμοποιείται [2,3,4] για να περιγράψει την εξέλιξη του παγκόσμιου ιστού από την σημερινή του μορφή, σε μια νέα, η οποία βασίζεται στις τεχνολογίες που περιγράψαμε νωρίτερα.

Συγκεκριμένα είναι γεγονός ότι το Web δεν έχει σχεδιαστεί για την χρήση την οποία υφίσταται. Αρχικά χρησιμοποιήθηκε για την διανομή στατικών κειμένων μεταξύ επιστημονικών κοινοτήτων. Ο μόνος τρόπος αλληλεπίδρασης με τα υπερκείμενα ήταν με την επιλογή υπερσυνδέσμων οι οποίοι οδηγούσαν σε άλλα κείμενα κόκ. Το είδος αυτής της αλληλεπίδρασης υλοποιούνταν πλήρως από το πρωτόκολλο HTTP. Από τα μέσα του 1995, όμως, άρχισε η εισαγωγή πιο δυναμικών μορφών αλληλεπίδρασης. Με τον όρο δυναμικών, εννοούμε αφενός την δυνατότητα νέων μορφών αλληλεπίδρασης από την πλευρά του browser, με χρήση DHTML ή ακόμα και Java applets, και αφετέρου την προσθήκη επιπλέον λειτουργιών στον web server εκτός από την προκαθορισμένη λειτουργία της παροχή έτοιμων σελίδων html. Εμείς θα αναφέρουμε περιληπτικά τις διάφορες επεκτάσεις από την πλευρά του server.

Η πρώτη και πιο σημαντική επέκταση του HTTP ήταν η προσθήκη ενός πρωτοκόλλου του *CGI (Common Gateway Interface)* που επέτρεπε στον web server να μεταβιβάσει μια request από κάποιον browser σε ένα πρόγραμμα το οποίο θα ήταν υπό την επίβλεψη του. Το συγκεκριμένο πρόγραμμα θα έκανε συνήθως μια αναζήτηση σε κάποια βάση δεδομένων και θα δημιουργούσε δυναμικά μια νέα σελίδα την οποία και θα έστελνε στον browser ο οποίος έκανε την αίτηση. Ο συνδυασμός HTTP/CGI δουλεύει αρκετά καλά, έχει όμως ένα πολύ σημαντικό μειονέκτημα, καθώς το HTTP δεν μπορεί να διατηρεί πληροφορίες κατάστασης για κάθε client που συνδέεται μαζί του. Κάθε νέα αίτηση για μια σελίδα *html* είναι άσχετη από την προηγούμενη. Το πρόβλημα αυτό έχει παρακαμφθεί με νέες τεχνικές όπως τα cookies, που είναι κάποια δεδομένα τα οποία αποθηκεύονται στον client εκ μέρους ενός Web Server. Επιπλέον το πρωτόκολλο CGI είναι αργό καθώς κάθε νέα request εξυπηρετείται από καινούρια διεργασία και έτσι το σύστημα υφίσταται τον φόρτο που σχετίζεται με την εκκίνηση της. Το παραπάνω μοντέλο υπέστη αρκετές επεκτάσεις με διάφορους μηχανισμούς όπως τα *Java Servlets,* οι *Active Server Pages (ASPs) κά.* Παρόλο που οι επεκτάσεις αυτές έλυσαν κάποια προβλήματα μπορούν να θεωρηθούν ώς προσωρινές λύσεις. Το βασικό πρόβλημα παραμένει, και είναι όπως προαναφέρθηκε, ότι το Web δεν σχεδιάστηκε για την χρήση που του γίνεται σήμερα.



Σχήμα 6. Τα συστατικά του Object Web

Το Object Web χρησιμοποιεί τις τεχνολογίες που περιγράψαμε νωρίτερα. Η λειτουργία του φαίνεται στο παραπάνω σχήμα και περιγράφεται παρακάτω:

Ο κάθε χρήστης χρησιμοποιώντας έναν web browser σαν αυτούς που κυκλοφορούν σήμερα, θα μπορεί να επισκέπτεται ένα web site της επιλογής του, κατά τα γνωστά. Εκεί ο web server θα του στέλνει μέσω του πρωτοκόλλου HTTP μία σελίδα HTML η οποία θα έχει ενσωματωμένο ένα Java Applet. Εδώ σταματά και η χρήση του HTTP. Μόλις αρχίσει η εκτέλεση του applet από την Java VM του browser[3] αυτό θα ενεργοποιεί το ORB το οποίο με την σειρά του θα συνδέεται μέσω του πρωτοκόλλου IIOP με το ORB στην μεσαία ζώνη. Το εκεί ORB θα επικοινωνεί με κάποιον application server. Με τον τρόπο αυτό το applet συνδέεται με τον application sever. Ο application server τότε θα μπορεί να έχει πρόσβαση σε μία ΒΔ κάνοντας διάφορες λειτουργίες και στέλνοντας τα όποια αποτελέσματα μέσω του IIOP στο applet, ώστε να φανούν στον χρήστη. Προαιρετικά μπορούν να παράγονται δυναμικά και σελίδες HTML οι οποίες να φαίνονται από τον browser του χρήστη με χρήση τότε πάλι του HTTP. Κάτι τέτοιο όμως δεν συμφέρει, καθώς η Java ως γλώσσα προγραμματισμού προσφέρει πολλές επιπλέον δυνατοτότητες από την HTML για ένα φιλικό GUI. Εκτός από αυτό όμως η μέθοδος αυτή ενδείκνυται και από τεχνικής πλευράς, καθώς το HTTP/IIOP είναι πιο αποτελεσματικό από το HTTP/CGI και τις διάφορες άλλες επεκτάσεις του HTTP. Αυτό συμβαίνει επειδή μια από τις αιτίες για τα προβλήματα του HTTP/CGI είναι η εξής: Συνήθως στην μεσαία ζώνη της εφαρμογής έτσι κι αλλιώς υπάρχουν αντικείμενα τα οποία επικοινωνούν με την ΒΔ. Το πρόβλημα προκύπτει καθώς τα αποτελέσματα των ενεργειών των αντικειμένων αυτών πρέπει να μεταφερθούν στον client διαμέσου του web server με την μορφή ιστοσελιδων. Αυτή η παράκαμψη δεν υπάρχει λόγος να γίνει. Στο Object Web ο application server επικοινωνεί κατευθείαν με τον client (που μπορεί να είναι κανονική

[3] Ο Netscape Communicator από την έκδοση 4.04 μαι μετά ενσωματώνει ένα από τα πιο δημοφιλή ORBs το VisiBroker της εταιρείας Inprise. Σε όλους τους άλλους browsers τα bytecodes του ORB πρέπει να τα στελνει ο web server όπως ακριβώς κάνει και με τα bytecodes του applet.

εφαρμογή) διαμέσου του IIOP. Η νέα αυτή προσέγγιση έχει διερευνηθεί ότι έχει καλύτερη απόδοση από τις προηγούμενες στο [5]. Τα πλεονεκτήματα μπορούν να συνοψισθούν ως εξής:

- **Παράγεται λογισμικό με μεγαλύτερη και ευκολότερη συντηρησιμότητα.** Πραγματικά οι νέες εκδόσεις μιας εφαρμογής τοποθετούνται απλώς στην θέση του παλιού applet στον web server. Η διανομή κώδικα και η εγκατάσταση είναι αυτόματη.
- **Καλύτερο User Interface.** Επειδή τα applets αναπτύσσονται με χρήση μιας πλήρους γλώσσας προγραμματισμού και όχι μόνο με την χρήση scripting languages. Έτσι μπορεί να υπάρχει περισσότερη οικειότητα με το user interface καθώς δεν διαφέρει από αυτά των παραδοσιακών εφαρμογών καθώς επίσης και περισσότερη λειτουργικότητα λόγω των πιο πλούσιων γραφικών στοιχείων.
- **Μεγαλύτερη ταχύτητα της εφαρμογής.** Επιτυγχάνεται από συνδυασμό πολλών παράγοντων όπως για παράδειγμα ότι δεν απασχολείται ο web server για να μεταφέρει περισσότερα documents, το IIOP είναι ταχύτερο από HTTP και το user interface συγκεντρώνει περισσότερη λειτουργικότητα.

Η προσέγγιση αυτή έχει βέβαια και αρκετά μειονεκτήματα τα οποία προέρχονται κυρίως από εξωγενείς παράγοντες. Το πιο σημαντικό το οποίο έχουμε ήδη προαναφέρει είναι τα προβλήματα των browsers σε ότι αφορά τις Java VM που αυτοί ενσωματώνουν. Για παράδειγμα σήμερα ο *Internet Explorer 5*, ο πιο δημοφιλής browser, υποστηρίζει Java 1.1 (- και μάλιστα ένα υποσύνολο της), η οποία είναι 2 χρόνια πίσω από τις σημερινές JVM. Μία λύση σε αυτό το πρόβλημα προσπάθησε να δώσει η Sun, δίνοντας την δυνατότητα στα applets να χρησιμοποιούν την τρέχουσα JVM μέσω ενός plug in. Αυτό όμως είναι ενα σχετικά μεγάλο αρχείο, το ‘κατέβασμα’ του οποίου είναι μια χρονοβόρα διαδικασία ιδιαίτερα για αργές dial – up συνδέσεις.

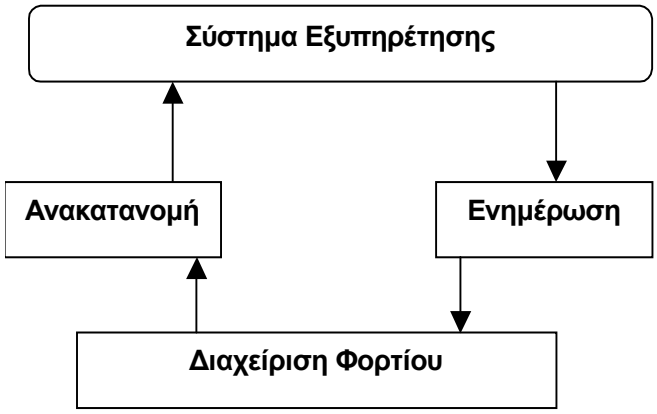
Τέλος αξίζει να αναφερθεί ότι κ οι προσπάθειες για την καθιέρωση του Object Web δεν προέρχονται μόνμο από τον OMG. Και η Microsoft φτιάχνει το δικό της μοντέλο που βασίζεται στην χρήση του DCOM αντί για το CORBA. Στην πλευρά του client όμως έχουμε κυριώς DHTML. Η εφαρμογή του όμως περιορίζεται σε πλατφόρμες που βασίζονται στα Windows.

3. ΔΙΑΧΕΙΡΙΣΗ ΦΟΡΤΙΟΥ

3.1 Γενικές Έννοιες

Όπως προαναφέρθηκε στην εισαγωγή, το πρόβλημα της κατανομής φορτίου σε ένα περιβάλλον client – server προκύπτει εξαιτίας της άνισης κατανομής αιτήσεων εξυπηρέτησης σε σύνολο από διαθέσιμους εξυπηρετητές. Έτσι ορισμένοι servers είναι υπερφορτωμένοι, ενώ άλλοι δεν χρησιμοποιούνται. Η κατανομή φορτίου έχει ως στόχο την διανομή των αιτήσεων στους εξυπηρετητές έτσι ώστε να βελτιωθεί η συνολική απόδοση του συστήματος. Για τον σκοπό αυτόν μπορούν να χρησιμοποιηθούν δύο τεχνικές: Η *Εξισορρόπηση Φορτίου (Load Balancing),* η οποία έχει ως σκοπό να χωρίσει εξίσου το φορτίο στους διαθέσιμους εξυπηρετητές και ο *Διαμοιρασμός Φορτίου (Load Sharing)* που έχει ως σκοπό να αφαιρέσει φορτίο από τους υπερφορτωμένους εξυπηρετητές και να το μεταφέρει στους λιγότερο φορτωμένους. Είναι φανερό ότι στην δεύτερη περίπτωση, το συνολικό φορτίο στο σύστημα δεν θα είναι εξίσου κατανεμημένο.

Κάθε σύστημα κατανομής φορτίου μπορεί να περιγραφεί με το παρακάτω σχήμα. Το σύνολο των εξυπηρετητών των οποίων το φορτίο παρακολουθείται, ενημερώνει συνεχώς το σύστημα διαχείρισης φορτίου. Το τελευταίο εκτιμά κάθε φορά τα νέα δεδομένα και αποφασίζει πως θα κατανέμει τις νέες αιτήσεις, έτσι ώστε να ικανοποιηθούν οι προκαθορισμένες απαιτήσεις απόδοσης.



Σχήμα 7. Σύστημα Διαχείρισης Φορτίου

Λέγοντας φορτίο εννούμε τις διάφορες αιτήσεις που υποβάλλονται στους εξυπηρετητες για επεξεργασία. Ο παραπάνω όρος είναι κάπως αφηρημένος. Στην πραγματικότητα το φορτίο όπως το αντιλαμβάνεται το σύστημα είναι οι τιμές ορισμένων μεγεθών για κάθε εξυπηρετητή μας. Η διαχείριση του φορτίου γίνεται με βάση τις τιμές αυτών των μεγεθών για κάθε εξυπηρετητή. Τα μεγέθη που χρησιμοποιούνται συνήθως για την μέτρηση του φορτίου είναι:

- Ο χρόνος απόκρισης των εξυπηρετητών.

- Το μέγεθος της ουράς αναμονής των εξυπηρετητών, δηλαδή το πλήθος των αιτήσεων που έχουν υποβληθεί από τους διάφορους πελάτες.
- Το ποσοστό χρησιμοποίησης του κάθε εξυπηρετητή, δηλαδή ο χρόνος ο οποίος ένας συγκεκριμένος εξυπηρετητής χρησιμοποιείται για την επεξεργασία μιας αίτησης.

3.2 Στρατηγικές Κατανομής Φορτίου

Η στρατηγική κατανομής φορτίου είναι απλά ο αλγόριθμος ο οποίος χρησιμοποιείται για να επιλέξουμε τον επόμενο εξυπηρετητή, τον επόμενο αποδέκτη μιας αίτησης δηλαδή. Μπορεί να ειπωθεί ότι κάθε τέτοιος αλγόριθμος αποτελείται από 4 βασικές διαδικασίες:

- Διαδικασία Πληροφόρησης (*Information Policy*): Ορίζει τι είδους πληροφορία πρέπει να συλλέγεται από το όλο σύστημα, πότε πρέπει αυτή να συλλέγεται και από ποια συστατικά στοιχεία. Για παράδειγμα μέρος της διαδικασίας πληροφόρησης είναι ο καθορισμός των μεγεθών που συνιστούν την έννοια του φορτίου για μια συγκεκριμένη περίπτωση. Επίσης μέρος της διαδικασίας πληροφόρησης είναι το αν οι τιμές των παραπάνω μεγεθών συλλέγονται περιοδικά, είτε όποτε είναι αυτές απαραίτητες.
- Διαδικασία Μεταφοράς (*Transfer Policy*) : Ορίζει πότε πρέπει να γίνει μια απόφαση κατανομής φορτίου. Η συγκεκριμένη διαδικασία βασίζεται συνήθως σε κάποιο όριο – κατώφλι. Όταν αυτό ξεπεραστεί (πχ. *server utilization > 70%*) σημαίνει ότι ο συγκεκριμένος εξυπηρετητής είναι υπερφορτωμένος και πρέπει να γίνει μια αλλαγή στον τρόπο κατανομής του φορτίου.
- Διαδικασία Επιλογής (*Selection Policy*) : Είναι αυτή που αποφασίζει ποια αίτηση θα μεταφερθεί όταν κάτι τέτοιο κριθεί απαραίτητο από την διαδικασία μεταφοράς. Στην διαδικασία επιλογής υπάρχουν δύο δυνατότητες. Πρώτον, να μεταφέρονται μόνο νέες αιτήσεις, αιτήσεις δηλαδή που δεν έχουν αρχίσει ακόμα να εξυπηρετούνται ή εναλλακτικά να μεταφέρονται και νέες αιτήσεις και αιτήσεις που εξυπηρετούνται ήδη. Η δεύτερη δυνατότητα είναι αρκετά πολύπλοκη καθώς εκτός από την ίδια την αίτηση πρέπει να μεταφέρεται και η κατάσταση της. Η πρώτη δυνατότητα είναι αρκετά απλή και για τον σκοπό αυτό είναι και η πιο συχνά χρησιμοποιούμενη.
- Διαδικασία Εντοπισμού (*Location Policy*) : Όταν πρέπει να μεταφερθεί μια αίτηση εντοπίζει τον εξυπηρετητή που θα την αποδεχθεί.

Με βάση τον τρόπο με τον οποίο κάθε στρατηγική υλοποιεί κάθε μία από τις παραπάνω διαδικασίες, μπορούμε να έχουμε τις εξής κατηγορίες στρατηγικών:

- *Στατικές – Δυναμικές* : Στις στατικές στρατηγικές, οι αποφάσεις για την κατανομή του φορτίου λαμβάνονται πριν την λειτουργία του συστήματος και δεν εξαρτώνται από την τρέχουσα κατάσταση του. Αντίθετα οι δυναμικές παρακολουθούν την κατάσταση του συστήματος και λαμβάνουν την απόφαση με βάση τις τιμές των μεγεθών μέτρησης του φορτίου. Έτσι, για παράδειγμα, μια στατική στρατηγική μπορεί να στείλει μια αίτηση σε έναν ήδη απασχολημένο εξυπηρετητή παρόλο που μπορεί να υπάρχει και κάποιος ελεύθερος. Κάτι τέτοιο δεν θα γίνει ποτέ με μια δυναμική στρατηγική.

- *Συγκεντρωτικές - Αποκεντρωτικές* : Στις συγκεντρωτικές στρατηγικές οι αποφάσεις για την διαχείριση του φορτίου λαμβάνονται από ένα κεντρικό συστατικό της εφαρμογής, ενώ στις αποκεντρωτικές η διαδικασία λήψης αποφάσεων είναι διεσπαρμένη σε όλο το δίκτυο. Ενώ οι πρώτες είναι αρκετά απλές στην υλοποίηση παρουσιάζουν προβλήματα γιατί η ανάθεση των αποφάσεων σε ένα μόνο συστατικό του δημιουργεί επιπρόσθετο φορτίο.

- *Sender Initiated – Receiver Initiated* : Εδώ η διάκριση γίνεται με βάση το αντικείμενο που ξεκινά την μεταφορά μιας αίτησης. Αν αυτό γίνεται από τους υπερφορτωμένους εξυπηρετητές έχουμε *sender – initiated*, καθώς αυτοι προσπαθούν να διώξουν το φορτίο από πάνω τους. Αν γίνεται από τους λιγότερο φορτωμένους εξυπηρετητές έχουμε *receiver - initiated* στρατηγική.

Όλες οι παραπάνω κατηγορίες στρατηγικών κατανομής φορτίου έχουν ορισμένα κοινά στοιχεία, τα οποία τις περισσότερες φορές αφορούν διάφορα προβλήματα που αντιμετωπίζουν. Το πρώτο πρόβλημα είναι ότι το ίδιο το σύστημα κατανομής φορτίου παράγει φορτίο, το οποίο επιβαρύνει το σύστημα το οποίο ελέγχει. Αυτό συμβαίνει καθώς γίνεται ανταλλαγή πληροφορίας (τιμές των μεγεθών παρακολούθησης φορτίου) για να υπάρχει συνεχής ενημέρωση για την κατάσταση του συστήματος, κάτι που δεν θα συνέβαινε αν δεν υπήρχε το σύστημα κατανομής φορτίου. Ένα άλλο κοινό πρόβλημα που πρέπει να αντιμετωπιστεί είναι η περίοδος ενημέρωσης του συστήματος, έτσι ώστε οι πληροφορίες με βάση τις οποίες θα γίνεται η κατανομή φορτίου να είναι σύμφωνες με την κατάσταση του συστήματος και να μην αντανακλούν παλαιότερα στιγμιότυπα του. Η χρήση μικρών περιόδων ενημέρωσης οδηγεί σε σωστές αποφάσεις, ταυτόχρονα όμως εντείνει το πρώτο πρόβλημα καθώς ανταλλάσσονται περισσότερα μηνύματα. Αντιστρόφως, αν έχουμε μεγάλες περιόδους ενημέρωσης, τότε η κατάσταση του συστήματος μπορεί να έχει μεταβληθεί στον χρόνο που μεσολαβεί και η απόφαση κατανομής φορτίου να μην είναι σωστή με βάση τα πραγματικά δεδομένα, παράγεται όμως μικρότερο ποσό πλεονάζουσας κίνησης.

3.3 Υλοποιήσεις Συστημάτων Κατανομής Φορτίου

Στην ενότητα αυτή θα μελετηθούν ορισμένες υλοποιήσεις συστημάτων κατανομής φορτίου, οι οποίες μελετώνται στο [1] και χρησιμοποιούνται για το υπό ανάπτυξη σύστημα. Όπως

προαναφέρθηκε οι συγκεκριμένες μέθοδοι έχουν χρησιμοποιηθεί σε περιβάλλον WWW για την κατανομή των αιτήσεων που αφορούν ιστοσελίδες. Επίσης οι μέθοδοι αυτοί εφαρμόζουν τις προτεινόμενες τεχνικές στο επίπεδο δικτύου, με αποτέλεσμα να ενδιάφερονται μόνο για το πλήθος των πακέτων και άλλα σχετικά χαρακτηριστικά και όχι για το περιεχόμενο τους. Οι τεχνικές αυτές υλοποιούνται σε διάφορα εμπορικά προϊοντα είτε σε επίπεδο hardware (όπου ενσωματώνονται σε μεταγωγείς), είτε σε επίπεδο software όπου είναι είτε τμήμα Web Server, είτε τμήμα κάποιου λειτουργικού συστήματος. Στις περισσότερες περιπτώσεις όμως , η γενική ιδέα πάνω στην οποία βασίζονται μπορεί να εφαρμοστεί και για την κατανομή αντικειμένων σε περιβάλλον Object Web.

I. Η Round Robin Domain Name Server (RR-DNS). [6]

Η μέθοδος αυτή είναι από τις πρώτες που εφαρμόστηκαν για την κατανομή των αιτήσεων σε περιβάλλον web.

Τα βασικά στοιχεία της μεθόδου αυτής είναι:

Αντί για έναν web server το site αποτελείται από πολλούς http servers, οι οποίοι είναι ταυτόσημοι. Για να μην αντιγράφονται όλες οι ιστοσελίδες σε κάθε έναν από αυτούς η πρόσβαση γίνεται μέσω ενός διαμοιραζόμενου συστήματος αρχείων (στην συγκεκριμένη περίπτωση χρησιμοποιείται το Andrew File System).

Η κατανομή των http requests στους ταυτόσημους web servers γίνεται με χρήση της μεθόδου Round Robin DNS, κατα την οποία ο Domain Name Resolver λειτουργεί ως νοητός δρομολογητής. Αναλυτικότερα:

Όλοι οι web browsers βλέπουν τους πολλαπλούς HTTP servers με ένα όνομα. (www.xxx.yyy). Κάθε φορά που έχουμε μια αίτηση http για μια ιστοσελίδα ο Domain Name Server αναθέτει σε διαφορετικό http server την εξυπηρέτηση. Όταν οι διαθέσιμοι servers εξαντληθούν ξαναξεκινάμε από την αρχή της λίστας κ.ο.κ.

Μερικά από τα προβλήματα της μεθόδου αυτής είναι:

- Ενδεχομένως υπάρχουν πολλοί Name Servers μεταξύ του client και του σημείου που εφαρμόζεται ο RR-DNS, οι οποίοι αποθηκεύουν την διεύθυνση IP που προκύπτει από την ανάλυση του ονόματος (caching) Τότε η κατανομή φορτίου δεν θα ισχύσει στις επόμενες αίτησεις γιατι το RR-DNS θα παρακαμφθεί. Μία προτεινόμενη λύση σε αυτό το σημείο είναι ότι κάθε όνομα που έχει αναλυθεί (*resolver*) μπορεί να αποκτήσει μία τιμή *TTL (Time To Live)*, η οποία όταν λήξει δεν επιτρέπει την παράκαμψη του RR-DNS.
- Το ίδιο πρόβλημα συμβαίνει και όταν οι clients αποθηκεύουν οι ίδιοι την προκύπτουσα διεύθυνση IP.
- Η δρομολόγηση γίνεται στο επίπεδο IP. Κατα συνέπεια δεν μπορεί να υπάρχει διαφοροποίηση υπηρεσιών με βάση το περιεχόμενο των πακέτων.

- Αν κάποιος server καταρρεύσει θα πρέπει να τροποποιηθεί ο αλγόριθμος, ώστε να μην τον υπολογίζει στους διαθέσιμους.
- Επίσης άνιση κατανομή φορτίου μπορεί να προκύψει από την παρατήρηση ότι οι clients δεν κάνουν μεμονωμένες αιτήσεις αλλά εκρηκτικές (burst) καθώς μια ιστοσελίδα συνήθως περιέχει πολλά επιπλέον αντικείμενα.(εικόνες κτλ.).

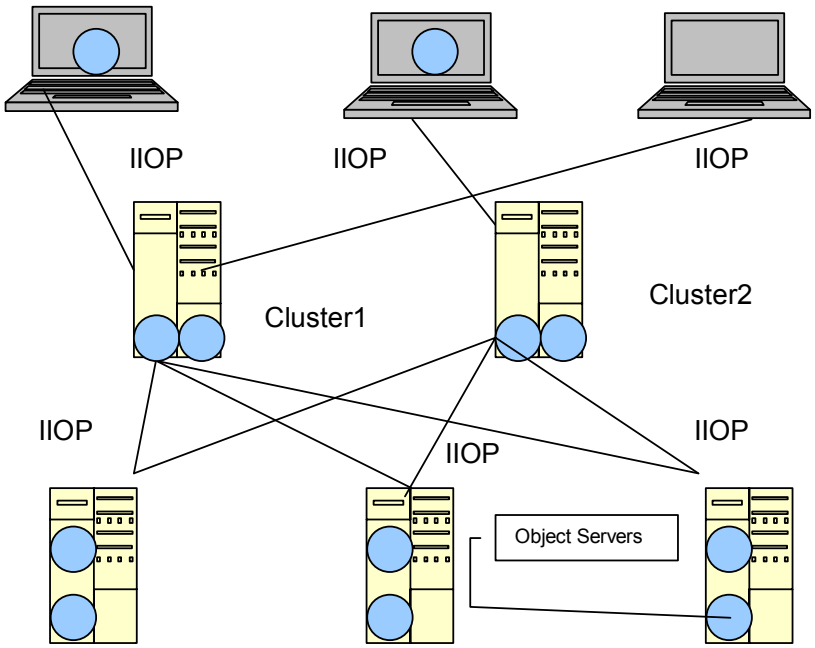
II. Παραλλαγή της μεθόδου I. [7]

Και εδώ οι διάφοροι web servers έχουν οργανωθεί σε αρχιτεκτονική συστοιχίας (cluster). Οι διάφοροι κόμβοι διακρίνονται όμως, λογικά, σε front - end και back - end. Οι πρώτοι συνδέονται απευθείας με το εξωτερικό δίκτυο και χρησιμοποιούνται κυρίως για κατανομή των εισερχόμενων αιτήσεων, ενω οι δεύτεροι χρησιμοποιούνται κυρίως για αποθήκευση. Οι web servers εκτελούνται εκεί. Οι τελευταίοι διαμερίζονται ανάλογα με το είδος των δεδομένων που περιέχουν. Όπως προαναφέρθηκε η παραπάνω διάκριση των κόμβων είναι λογική. Για την φυσική υλοποίηση προτείνονται 2 δυνατότητες: Μπορούν δηλαδή οι front end nodes να βρίσκονται στο ίδιο μηχάνημα με τους back - end (flat) ή σε διαφορετικό (2-tier).

Πριν από τους front end nodes υπάρχει ένας δρομολογητής, ο οποίος κατανέμει τις αιτήσεις με χρήση Round Robin DNS ή με κάποια άλλη μέθοδο στους front end nodes. Αυτοί τις εξυπηρετούν με τα δεδομένα που έχουν αποθηκευθεί στους back-end. Όταν είναι να σταλεί η απάντηση τα πακέτα λαμβάνουν την διεύθυνση του δρομολογητή, ώστε η όλη διαδικασία να είναι διαφανής στον client. Έτσι ξεπερνιέται το πρόβλημα της προηγούμενης προσέγγισης, όπου η αποθήκευση των διευθύνσεων IP παρέκαμπτε τον αλγόριθμο διανομής φορτίου. Εδώ όλα τα πακέτα έχουν την διεύθυνση του δρομολογητή. Επίσης, λύνεται το πρόβλημα της ‘έκρηξης αιτήσεων’ (burst) γιατί ο router κατανέμει κάθε request σε διαφορετικό front end node. Η συγκεκριμένη προσέγγιση μπορεί να εφαρμοστεί όχι μόνο για Web Servers, αλλά και για Video Servers.

Για λόγους πληρότητας, αξίζει να αναφέρουμε πως για την κατανομή φορτίου σε περιβάλλον Web έχουν προταθεί και άλλες αρχιτεκτονικές όπως για παράδειγμα ο διαχωρισμός των ιστοσελίδων ανάλογα με το περιεχόμενο τους και η χρήση ενός redirection server ο οποίος ανακατευθύνει την αρχική αίτηση στον web server που πραγματικά περιέχει την σελίδα. Ακόμα έχουν προταθεί και πιο ‘εξωτικές’ τεχνικές που τροποποιούν τους συνδέσμους των ιστοσελίδων ανάλογα με το φορτίο κάθε ενός από τους Web Servers, κάτι που είναι αποτελεσματικό με την προϋπόθεση ότι το site έχει προκαθορισμένα ‘σημεία εισόδου’. Το θέμα όμως της εργασίας δεν είναι αυτό. Οι παραπάνω αναφορές έγιναν καθώς η συγκεκριμένη εφαρμογή εφαρμόζει παραλλαγές των παραπάνω τεχνικών όχι πια για ιστοσελίδες, αλλά για αντικείμενα.

Η συγκεκριμένη εργασία βασίζεται στην γενική ιδέα της αρχιτεκτονικής που προτείνεται στον RR-DNS. Μπορεί να παρασταθεί με το παρακάτω σχήμα:



Σχήμα 8. Γενική Αρχιτεκτονική Εφαρμογής

Τονίζεται σχετικά με το παραπάνω ότι τα διάφορα αντικείμενα- εξυπηρετητές μπορούν να βρίσκονται και όλοι στον ίδιο υπολογιστή. Βασικά η φυσική τους τοποθεσία δεν έχει σημασία. Αυτά όμως θα περιγραφούν καλύτερα στην επόμενη ενότητα, όπου θα γίνει η περιγραφή της αρχιτεκτονικής της εφαρμογής. Επίσης αξίζει να σημειωθεί τώρα ότι η εφαρμογή μπορεί να λειτουργήσει με χρήση παραπάνω από ενός cluster, τα οποία όμως θα ελέγχουν το ίδιο σύνολο αντικειμένων.

4. ΥΛΟΠΟΙΗΣΗ

Η συγκεκριμένη εργασία βασίζεται στο [1]. Εκεί παρουσιάζεται μία αρχιτεκτονική για διαχείριση φορτίου σε κατανεμημένα συστήματα, που ονομάζεται **COLD Java (CORBA Object Load Distributor in Java)** και χρησιμοποιεί τις τεχνολογίες που περιγράψαμε νωρίτερα, για την κατανομή αντικειμένων σε (ετερογενή) δίκτυα με απώτερο σκοπό την βελτίωση της συνολικής απόδοσης ενός συστήματος που παρέχει κάποια υπηρεσία. Το COLD Java λοιπόν, αποτελείται από ένα σύνολο από Java Interfaces, ένα CORBA ORB (Object Request Broker) και έναν Web Server. Ένα από τα κυριότερα χαρακτηριστικά του είναι ότι επιτρέπει την εκτέλεση λειτουργιών που κανονικά θα έπρεπε να εκτελούνται στον server host στον client host, χρησιμοποιώντας έτσι και την επεξεργαστική ισχύ που διακρίνει τα μεσαία PC σήμερα, για να διαμοιράσει το φορτίο του server της εφαρμογής. Στο COLD Java τα διάφορα αντικείμενα-εξυπηρετητές έχουν οργανωθεί σε συστοιχίες (clusters) σύμφωνα με τις μεθόδους εξισορρόπησης φορτίου που παρουσιάσαμε νωρίτερα. Ο client που στην συγκεκριμένη περίπτωση είναι ένα Java applet, 'κατεβάζει' μαζί του ένα αντικείμενο εξυπηρετητή, το οποίο ενεργοποιείται και συμμετέχει και αυτό στην ίδια συστοιχία με τα υπόλοιπα όμοια αντικείμενα, η οποία ελέγχεται από ένα αντικείμενο που βρίσκεται στον server host. Η μόνη διαφορά με τα υπόλοιπα μέλη του cluster είναι ότι αυτό που εκτελείται στον client 'γνωρίζει' ότι εκτελείται εκεί. Έτσι, όποια αποτελέσματα έχει, δεν τα στέλνει πίσω στον server (για να ξαναμεταφερθούν στον client), αλλά στέλνονται άμεσα στον client. Το cluster διαθέτει στοιχεία που του επιτρέπουν να παρακολουθεί το φορτίο όλων των αντικειμένων-εξυπηρετητών που διαχειρίζεται. Με βάση τα στοιχεία αυτά, μπορεί να ρυθμίζει την πολιτική επιλογής εξυπηρετητή για κάποια εργασία, έτσι ώστε η απόδοσή του συστήματος να είναι βέλτιστη. Όλη η αρχιτεκτονική του COLD Java εφαρμόζεται πρακτικά με μια υλοποίηση επίλυσης του *προβλήματος του 'πλανόδιου πωλητή'* (*Traveling Salesman Problem*).

Η εργασία αυτή δανείζεται την γενική ιδέα από το [1] και παρουσιάζει μια ανάλογη αρχιτεκτονική. Η αρχιτεκτονική που προτείνεται εδώ εφαρμόζεται πρακτικά, χρησιμοποιώντας RMI (υπάρχει και αντίστοιχη έκδοση για CORBA) σε μια εφαρμογή για την επίλυση του προβλήματος της εύρεσης του συντομότερου μονοπατιού σε ένα γράφημα, από κάθε κορυφή του προς όλες τις άλλες, χρησιμοποιώντας τον αλγόριθμο του Dijkstra. Στόχος αυτού του κεφαλαίου είναι η περιγραφή της συγκεκριμένης αρχιτεκτονικής. Θα χρησιμοποιήσουμε μία bottom-up προσέγγιση, ξεκινώντας από την μελέτη του προβλήματος, προχωρώντας στην περιγραφή της λειτουργίας της εφαρμογής και καταλήγοντας στην περιγραφή της γενικής αρχιτεκτονικής.

4.1 Ο αλγόριθμος του Dijkstra

Το πρόβλημα το οποίο επιλύει ο αλγόριθμος του Dijkstra (1959) είναι η εύρεση του συντομότερου μονοπατιού από μία κορυφή ενός κατευθυνόμενου γραφήματος σε όλες τις άλλες. Λέγοντας μονοπάτι εννούμε το σύνολο των τόξων που ξεκινούν από μία κορυφή και καταλήγουν σε μία άλλη. Προφανώς, ο συγκεκριμένος αλγόριθμος θεωρεί ότι κάθε μονοπάτι στο γράφημα έχει ένα συγκεκριμένο βάρος , με συνέπεια δύο μονοπάτια που ξεκινούν από την ίδια κορυφή και καταλήγουν πάλι σε ίδια κορυφή δεν είναι πάντα ισοδύναμα. Το συνολικό βάρος ενός μονοπατιού ορίζεται ως το άθροισμα των μεμονωμένων βαρών κάθε τόξου που αποτελεί το μονοπάτι. Ο αλγόριθμος του Dijkstra για να τερματίσει, απαιτεί το βάρος κάθε τόξου να είναι **θετικό** [13]. Τέλος έχει πολυπλοκότητα τάξεως O($|V|^2$) όπου |V| είναι ο αριθμός των κορυφών του γραφήματος, όπως άλλωστε μπορεί να δει κανείς και παρακάτω:

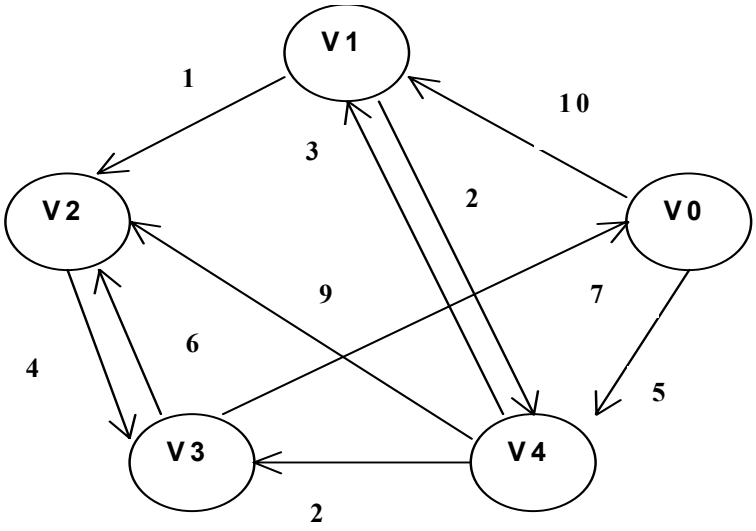
```
Begin
D[vo]    ←  0;
for each   v ∈ V − {v_o} do D[vo]    ←  ∞;
S ← ∅;
B ← {v_o};
 while B   <> ∅ do
   begin
      choose  a vertex v ∈ B such that   D[v]   =   min;
      S  ← S ∪ {v};
      B  ← B − {v};
       for each   u  ∈  V-S such   that (v,u)   ∈ E do
       begin
            D[u]   ← min {D[u],D[v]   + d_(v,u) };

            B  ← B ∪ {u};
        end
   end
end
```

Σχήμα 9 Ο αλγόριθμος του Dijkstra

Η συγκεκριμένη εργασία βασίζεται σε μια υλοποίηση του αλγορίθμου, η οποία περιγράφεται στο [14]. Στην υλοποίηση αυτή έχουν γίνει αρκετές τροποποιήσεις έτσι ώστε να καλυφθούν οι ανάγκες του συγκεκριμένου συστήματος.

Παρακάτω θα περιγράψουμε μια δοκιμαστική εκτέλεση του αλγορίθμου για ένα μικρό γράφημα.



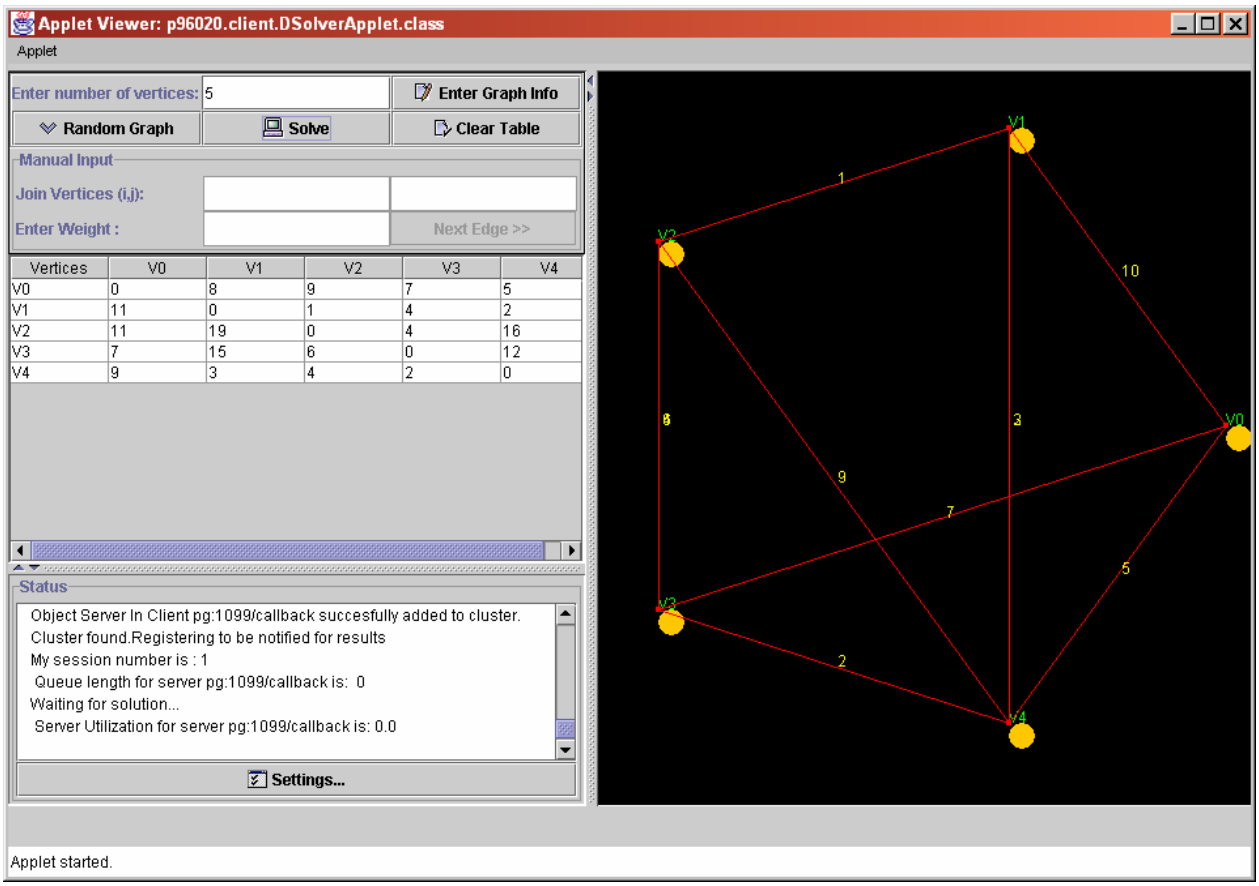
Στην παραπάνω εικόνα φαίνεται η αρχική μορφή του γραφήματος μας. Θέλουμε να βρούμε το ελάχιστο μονοπάτι από την κορυφή νο προς όλες τις άλλες κορυφές. Αρχικά όλοι οι κόμβοι έχουν επιγραφή (*D[v]*) άπειρο εκτός από το νο που έχει 0. Το σύνολο S είναι κενό και το σύνολο B περιέχει μόνο την κορυφή νο.

Επανάληψη 1:	Επιλέγουμε την κορυφή νο, την αφαιρούμε από το B και την προσθέτουμε στο S. Οι γειτονικές της κορυφές, τις οποίες δεν έχουμε επισκεφθεί, είναι οι v1, v4. Αυτές εισάγονται στο B και αποκτούν επιγραφή 10, 5 αντίστοιχα.
Επανάληψη 2:	Η κορυφή του B με την ελάχιστη περιγραφή είναι η **v4**. Την αφαιρούμε και την προσθέτουμε στο S. Οι γειτονικές της είναι οι v1, v2, v3. Αυτές προστίθενται στο B με νέες επιγραφές 8, 14, 7. **Το μικρότερο μονοπάτι από την v0 στην v4 έχει κόστος 5.**
Επανάληψη 3:	Η κορυφή του B με την ελάχιστη περιγραφή είναι η **v3**. Την αφαιρούμε και την προσθέτουμε στο S. Οι γειτονικές της είναι οι v0, v2. Η v2 προστίθεται στο B με νέα επιγραφή 13. **Το μικρότερο μονοπάτι από την v0 στην v3 έχει κόστος 7.**
Επανάληψη 4:	Η κορυφή του B με την ελάχιστη περιγραφή είναι η **v1**. Την αφαιρούμε και την προσθέτουμε στο S. Οι γειτονικές της είναι οι v2, v4. Η v2 προστίθεται στο B με νέα επιγραφή 9. **Το μικρότερο μονοπάτι από την v0 στην v1 έχει κόστος 8.**
Επανάληψη 5:	Η κορυφή του B με την ελάχιστη περιγραφή είναι η **v2** (η τελευταία). Την αφαιρούμε και την προσθέτουμε στο S. **Το μικρότερο μονοπάτι από την v0 στην v2 έχει κόστος 9.**

Τα υπόλοιπα αποτελέσματα για το γράφημα φαίνονται στο σχήμα 10, όπως υπολογίστηκαν από το πρόγραμμα.

4.2 Λειτουργία

Για να γίνει κατανοητή η αρχιτεκτονική της εφαρμογής-πρότυπο, καλό είναι να ρίξουμε μια ματιά πρώτα στον τρόπο λειτουργίας της. Η λειτουργία της εφαρμογής λοιπόν έχει ως εξής: Κάποιος επισκέπτεται ένα web site και ο web server του κατεβάζει μία σελίδα HTML η οποία περιέχει ένα Java applet. Το applet παρέχει τα συστατικά διεπαφής με τον χρήστη (*User Interface Components)* τα οποία θα του επιτρέψουν να εισάγει όλες εκείνες τις πληροφορίες που είναι απαραίτητες για την αναπαράσταση του γραφήματος. Επίσης το applet περιέχει και τα συστατικά εκείνα για την εμφάνιση των αποτελεσμάτων πίσω στον χρήστη. Τελικά λοιπόν παρουσιάζεται η εξής οθόνη:

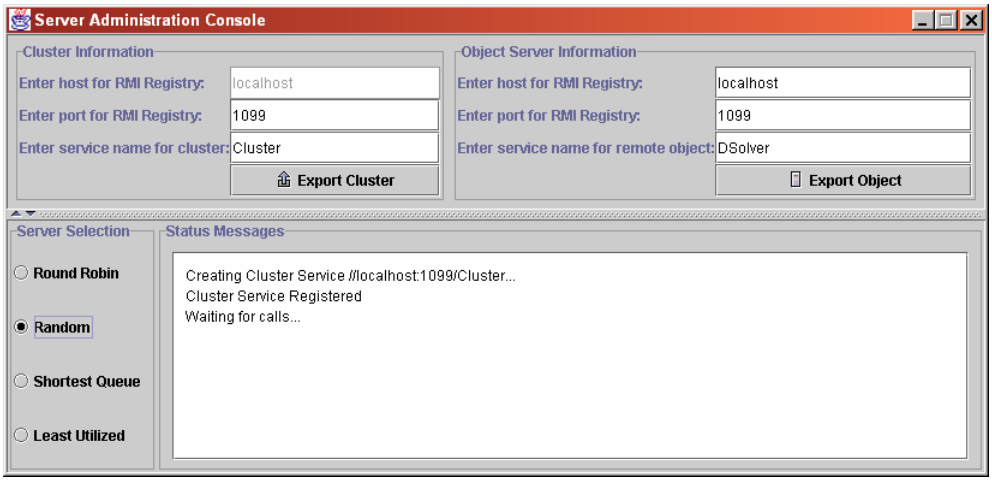


Σχήμα 10. Το applet για την εισαγωγή του γραφήματος

Για την εισαγωγή του γραφήματος στο σύστημα δίνονται δύο δυνατότητες. Πρώτα από όλα ο χρήστης μπορεί να εισάγει το γράφημα μόνος του ή εναλλακτικά μπορεί να αφήσει το πρόγραμμα να δημιουργήσει ένα τυχαίο γράφημα. Η δεύτερη επιλογή παρέχεται, καθώς η

λειτουργία της διαχείρισης του φορτίου φαίνεται καλύτερα στα μεγάλα γραφήματα, τα οποία είναι δύσκολο να δωθούν '*με το χέρι*'.

Το παραπάνω applet **δεν επιλύει** τον αλγόριθμο. Η επίλυση του αλγορίθμου γίνεται από μία σειρά από αντικείμενα - εξυπηρετητές τα οποία, μπορεί να βρίσκονται σε κάποιο τοπικό δίκτυο στο οποίο συμμετέχει και ο host όπου λειτουργεί ο web server που φιλοξενεί το applet. Η παραπάνω διάταξη βέβαια δεν είναι υποχρεωτική. Τα αντικείμενα εξυπηρετητές μπορεί να βρίσκονται σε οποιοδήποτε μηχάνημα έχει σύνδεση στο Internet. Στην συγκεκριμένη εφαρμογή για τον ρόλο του Web Server, έχει χρησιμοποιηθεί ο *Java Web Server 2.0*, κάτι όμως που δεν έχει ιδιαίτερη σημασία. Τα αντικείμενα εξυπηρετητές είναι τα αντικείμενα που αναλαμβάνουν την επίλυση του αλγορίθμου. Οργανώνονται σε μια αρχιτεκτονική cluster, όπου ελέγχονται από ένα αντικείμενο στο οποίο θα αναφερόμαστε από εδώ και στο εξής ως cluster. Μόλις λοιπόν ο χρήστης εισάγει τις παραμέτρους του γραφήματος, γίνεται η επικοινωνία του applet με το cluster. Για τον σκοπό αυτόν το cluster **πρέπει να εκτελεστεί στον ίδιο host με τον web server**, καθώς η Java VM απαγορεύει στα *untrusted applets* (όπως το συγκεκριμένο) να επικοινωνούν με άλλον host εκτός από αυτόν από τον οποίο φορτώθηκαν. Το μήνυμα που στέλνει το applet στο cluster, έχει ως αποτέλεσμα το πέρασμα του γραφήματος εκεί και την δήλωση του applet στο σύνολο με τους clients που διαχειρίζεται το cluster. Το cluster τότε ελέγχει πόσα αντικείμενα - εξυπηρετητές έχει στην διάθεση του και διαμέσου ενός βοηθητικού αντικειμένου κατανέμει τις αιτήσεις για επίλυση σε αυτά. Ο τρόπος με τον οποίο γίνεται η κατανομή μπορεί να επιλεγεί από τον χρήστη μέσα από τέσσερις εναλλακτικές λύσεις. Η πρώτη κατανέμει το φορτίο στα διάφορα αντικείμενα εξυπηρετητές χρησιμοποιώντας την μέθοδο Round Robin. Η δεύτερη δυνατότητα κατανέμει τις αιτήσεις με τυχαίο τρόπο. Τέλος υπάρχει και επιλογή για κατανομή μιας αίτησης στο αντικείμενο-εξυπηρετητή με την μικρότερη ουρά ή τον μικρότερο βαθμό χρησιμοποίησης αντίστοιχα. Όλα τα παραπάνω στοιχεία, καθώς επίσης και η δήλωση των αντικειμένων εξυπηρετητών στο cluster γίνονται με την βοήθειας της 'κονσόλας' που παρουσιάζεται στο παρακάτω σχήμα:



Σχήμα 11. 'Κονσόλα' Διαχείρισης Cluster.

Όπως προαναφέρθηκε, υποθέτουμε ότι τα διάφορα αντικείμενα εξυπηρετητές βρίσκονται σε ένα τοπικό δίκτυο μαζί με την εφαρμογή server, κάτι που δεν είναι απαραίτητο. Μετά την κατανομή των αιτήσεων δίνεται εντολή ταυτόχρονα σε όλους τους εξυπηρετητές να ξεκινήσουν την επίλυση του προβλήματος. Κάθε φορά που τελειώνουν μία κορυφή, δηλαδή βρίσκουν το ελάχιστο μονοπάτι που ξεκινά από αυτήν και καταλήγει σε οποιαδήποτε άλλη κορυφή του γραφήματος, επιστρέφουν τα αποτελέσματα στην server εφαρμογή, η οποία με την σειρά της τα στέλνει στον client, για να παρουσιαστούν στον χρήστη. Ταυτόχρονα με κάθε ενημέρωση για αποτελέσματα τα αντικείμενα στέλνουν ενσωματωμένη και την ενημέρωση για το φορτίο τους στο cluster. Εκεί υπάρχουν ειδικά αντικείμενα τα οποία συγκεντρώνουν τις πληροφορίες, ώστε να μπορεί να γίνει δυναμική προσαρμογή φορτίου, δηλαδή στην επόμενη αίτηση να ανατεθεί η επίλυση στο 'λιγότερο φορτωμένο'αντικείμενο. Το 'φορτίο' εκφράζεται με την βοήθεια δύο μεγεθών, σύμφωνα με το [1]:

- Το πλήθος των αιτήσεων που περιμένουν να εξυπηρετηθούν.
- Το ποσοστό του χρόνου που το κάθε αντικείμενο είναι απασχολημένο.

Στην παρούσα έκδοση δεν κατέστη δυνατή η δυναμική κατανομή του φορτίου με βάση τα παραπάνω μεγέθη, καθώς τα αποτελέσματα των παραπάνω τιμών τα οποία λαμβάνονται κάθε 10 δευτερόλεπτα είναι όλα 0, μη επιτρέπωντας έτσι την εξαγωγή κάποιου συμπεράσματος.

Μαζί με το applet όμως έρχεται και ένα αντικείμενο εξυπηρετητής. Ο χρήστης μπορεί να επιλέξει αν θα τον ενεργοποιήσει ή όχι. Αν τον ενεργοποιήσει τον δηλώνει κανονικά στο cluster. Η διαχείριση του συγκεκριμένου αντικειμένου από το cluster δεν διαφέρει από τα άλλα κατανεμημένα αντικείμενα. Η αρχικοποίηση, ο έλεγχος και η ενημέρωση για το φορτίο γίνονται ακριβώς με τον ίδιο τρόπο. Η μόνη διαφορά είναι στον τρόπο ενημέρωσης του client για τα αποτελέσματα. Αυτή γίνεται απευθείας και όχι μέσω του cluster, κάτι που οδηγεί σε λιγότερες κλήσεις μέσω δικτύου και άρα σε μεγαλύτερη αποδοτικότητα της εφαρμογής.

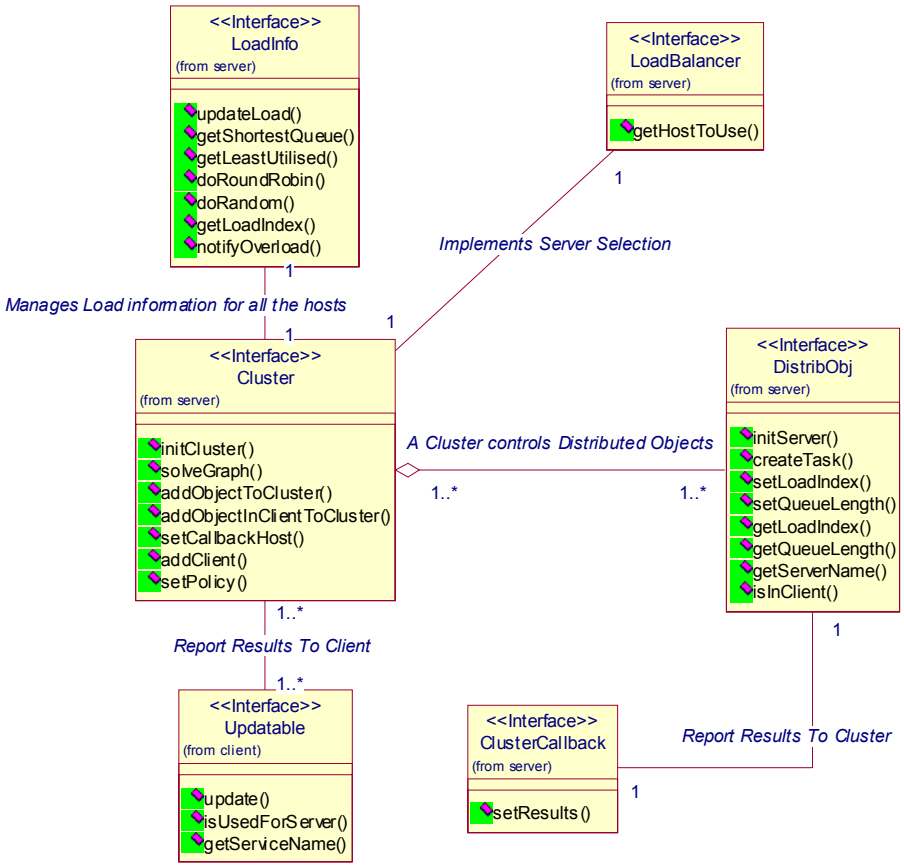
Αυτή ήταν σε γενικές γραμμές η λειτουργία της εφαρμογής. Θα προχωρήσουμε τώρα στην περιγραφή όλων των αντικειμένων που απαρτίζουν το σύστημα και των αλληλεπιδράσεων τους.

4.3 Προτεινόμενη Αρχιτεκτονική.

Παρακάτω θα περιγράψουμε την προτεινόμενη αρχιτεκτονική του συστήματος. Για τον σκοπό αυτόν θα χρησιμοποιηθεί ο συμβολισμός της UML *(Unified Modeling Language)* , η οποία είναι διεθνές πρότυπο στην περιγραφή αντικειμενοστρεφών συστημάτων. Τα διαγράμματα που παρουσιάζονται έχουν γίνει με χρήση του CASE εργαλείου **Rational Rose For Java.**

Θα ξεκινήσουμε την ανάπτυξη, περιγράφοντας τα συστατικά εκείνα του συστήματος τα οποία είναι ανεξάρτητα του προβλήματος, το οποίο χρησιμοποιείται για την επίδειξη του συστήματος, δηλαδή του αλγορίθμου του Dijkstra. Αυτά τα συστατικά τα οποία ενδεχομένως μπορούν να χρησιμοποιηθούν και στην προσαρμογή του συστήματος σε ένα άλλο πρόβλημα, φαίνονται στο παρακάτω σχήμα με την μορφή interfaces. Για εξοικονόμηση χώρου δεν φαίνονται τα ορίσματα των μεθόδων, τα οποία όμως θα περιγραφούν κανονικά.

Αρχικά πρέπει να επισημάνουμε ότι δύο από τα interfaces που φαίνονται στο σχήμα δεν είναι απαραίτητα. Αυτά είναι το Updatable και το ClusterCallback. Το πρώτο χρησιμοποείται για να ενημερωθεί ο client για τα αποτελέσματα του αλγορίθμου και να τα παρουσιάσει στον χρήστη, και το δεύτερο για να ενημερωθεί το cluster για τα ίδια αποτελέσματα και να στείλει στον client. Η ύπαρξη τους οφείλεται σε μία αδυναμία του RMI και του CORBA.



Σχήμα 12. Τα βασικά στοιχεία του συστήματος.

Συγκεκριμένα, τα πρωτόκολλα IIOP και JRMP του CORBA και RMI, επειδή σχεδιάστηκαν με στόχο να είναι όσο το δυνατόν πιο απλά, είναι **μονόδρομα.** Δηλαδή ο client μπορεί να καλέσει μία μέθοδο του server , αλλά ο server δεν μπορεί να καλέσει μια μέθοδο του client. Πολλές φορές όμως πρέπει οι servers να μπορούν να καλούν τους clients για να τους ενημερώσουν για κάποια αποτελέσματα, όπως εδώ άλλωστε. Η λειτουργία αυτή όμως δεν υποστηρίζεται άμεσα από τα παραπάνω πρωτόκολλα. Το παραπάνω μειονέκτημα μπορεί να ξεπεραστεί με την τεχνική των *callbacks.* Συγκεκριμένα το callback είναι ένα remote αντικείμενο το οποίο δημιουργεί η client εφαρμογή. Το αντικείμενο αυτό διαθέτει μεθόδους οι οποίες καλούνται από τον server, όταν θέλει να ενημερώσει την client εφαρμογή για κάποια νεα δεδομένα. Με τον τρόπο αυτό έχουμε ναι μεν μονόδρομη επικοινωνία μεταξύ client και server, με αντεστραμμένους ρόλους όμως . Η τεχνική του *callback* χρησιμοποιείται 2 φορές, μία για να ενημερωθεί ο client για τα αποτελέσματα του αλγορίθμου από το cluster και μία για να ενημερωθεί το cluster για τα αποτελέσματα του αλγορίθμου και το φορτίο για κάθε ένα από τα αντικείμενα εξυπηρετητές.

Κάθε αντικείμενο που υλοποιεί το interface *Updatable* θα είναι στην ουσία ένας client του cluster. Η μέθοδος *update(int vo,int[] results)* καλείται από το cluster κάθε φορά που έχει ολοκληρωθεί η επίλυση μιας κορυφής και έχει ως αποτέλεσμα την εμφάνιση στον χρήστη των συντομότερων μονοπατιών από την κορυφή vo προς κάθε άλλη κορυφή του γραφήματος, τιμές οι οποίες περιέχονται στον πίνακα results. Η μέθοδος *boolean isUsedForServer()* καλείται από το cluster για να διαπιστωθεί αν στον συγκεκριμένο client υπάρχει αντικείμενο εξυπηρετητής. Αυτό συμβαίνει καθώς πρέπει οι servers που έχουν 'κατεβεί' στους clients να διατηρούνται-ξεχωριστά από τους υπόλοιπους, γιατί δεν είναι υποχρεωτικό ότι όλοι οι clients θα έχουν κατεβάσει και θα χρησιμοποιούν τον server τους. Στην περίπτωση που η παραπάνω μέθοδος επιστρέφει true, η μέθοδος *String getServiceName()* επιστρέφει ένα λογικό όνομα του server το οποίο χρησιμεύει ως κλειδί για την αποθηκέυση της reference σε ένα hash table του cluster.

Η δεύτερη χρήση της τεχνικής των callbacks εφαρμόζεται μέσω του *ClusterCallback*. Αυτό διαθέτει μία μέθοδο την *setResults(int clientNum, int vo, int[] results, String serverName, boolean inClient, int queueLength, float loadIndex)* την οποία καλούν τα αντικείμενα – εξυπηρετητές για να ενημερώσουν το cluster. Τα ορίσματα *vo, results* έχουν την ίδια σημασία με πριν. Το όρισμα *clientNum* είναι ένας τρόπος ταυτοποίησης του πελάτη για τον οποίο ισχύουν τα αποτελέσματα αυτά, ενώ το *serverName* είναι το λογικό όνομα της reference του server. Τα άλλα δύο ορίσματα είναι οι τιμές του 'φορτίου' του συγκεκριμένου αντικειμένου - εξυπηρετητή.

Κάθε αντικείμενο - εξυπηρετητής υλοποιεί το interface *DistribObj*. Αυτό περιέχει μεθόδους για την αρχικοποίηση του αντικειμένου αλλά και για την έναρξη της διαδικασία επίλυσης του γραφήματος. Όπως προαναφέρθηκε κάθε αντικείμενο – εξυπηρετητής ελέγχεται από το cluster. Πιο συγκεκριμένα, μπορεί να ελέγχεται από πολλά cluster και βέβαια θα λαμβάνει αιτήσεις για πολλούς πελάτες. Είναι απαραίτητη λοιπόν η ύπαρξη κάποιου μηχανισμού, ώστε να εξασφαλιστεί ότι δεν 'μπερδεύονται' οι αιτήσεις από τα διάφορα clusters και τους διάφορους πελάτες. Η ταυτοποίηση γίνεται στο επίπεδο της αίτησης, δηλαδή κάθε αίτηση για εύρεση του μικρότερου μονοπατιού συνοδεύται από έναν αριθμό ο οποίος καθορίζει τον πελάτη που έκανε την αίτηση, και από μία reference για το callback που χρησιμοποιείται για την ταυτοποίηση και ενημέρωση του cluster για τα αποτέλεσματα. Αυτό το γεγονός φαίνεται στην μέθοδο *createTask(ClusterCallback clustercallback, int clientNum, int numOfVertices, java.util.Vector edges, int vo)* που δημιουργεί ένα αντικείμενο Task, το οποίο θα εκτελέσει την επίλυση του αλγορίθμου. Η μέθοδος *initServer(String serverName)* καλείται από το Cluster και χρησιμοποιείται για την αρχικοποίηση του server με λογικό όνομα το όρισμα serverName. Η μέθοδος *isInClient()* επιστρέφει μια Boolean τιμή για το αν το συγκεκριμένο αντικείμενο έχει 'κατεβεί' και φορτωθεί στον πελάτη. Υπάρχουν επίσης και

μέθοδοι για την ανάκτηση και παρακολούθηση των τιμών του φορτίου του συγκεκριμένου εξυπηρετητή.

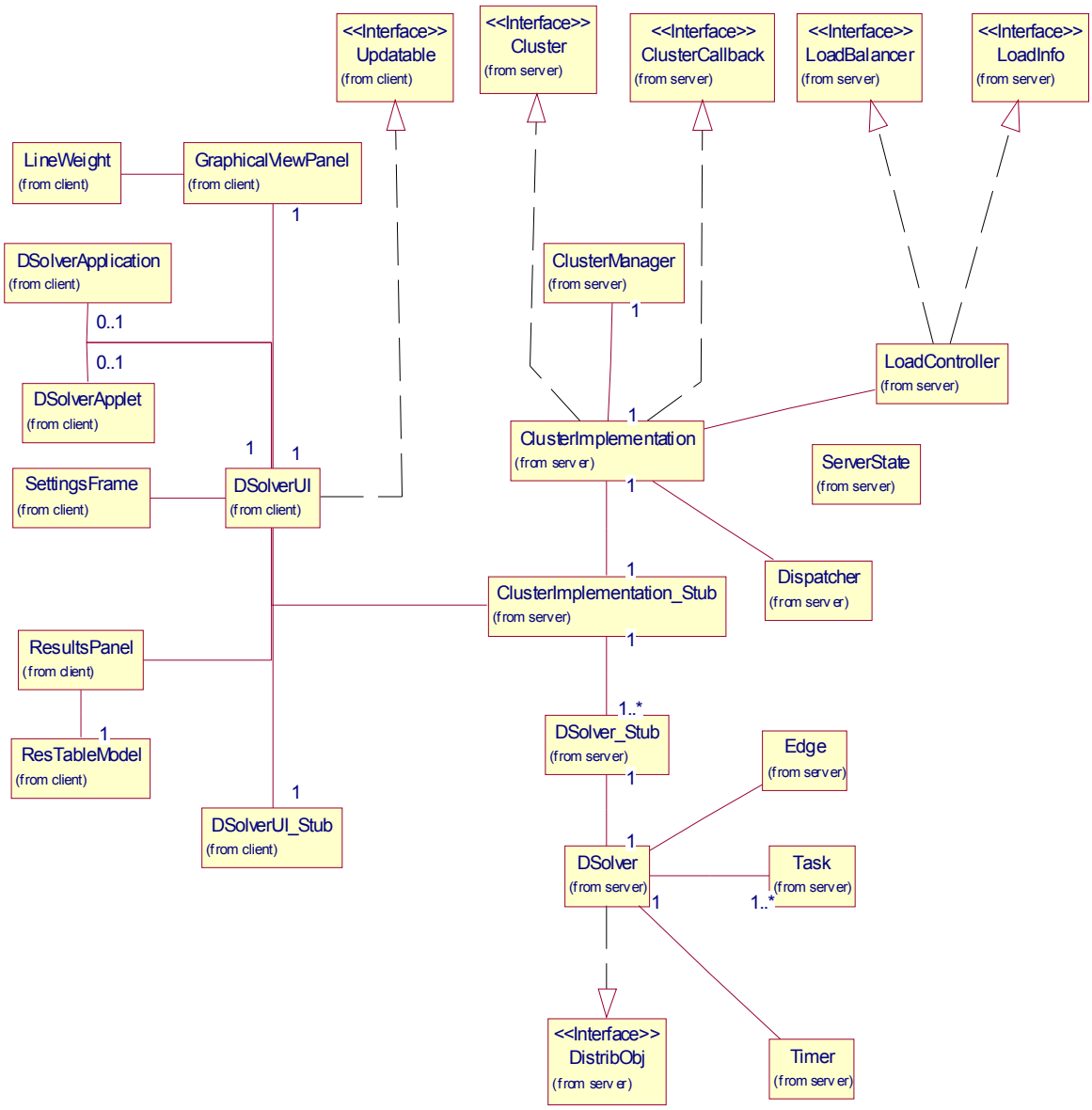
Το interface *Cluster* υλοποιείται από τα αντικείμενα τα οποία θα παίζουν τον ρόλο του cluster, θα διαχειρίζονται δηλαδή ένα σύνολο από αντικείμενα – εξυπηρετητές με ένα σύνολο από μεθόδους. Η μέθοδος *initCluster* λαμβάνει ως όρισμα το λογικό όνομα του cluster και λειτουργεί για την αρχικοποίηση του. Η μέθοδος *solveGraph* έχει ως αποτέλεσμα την κατανομή των αιτήσεων για την επίλυση του γραφήματος. Ως ορίσματα έχει τον αριθμό ταυτοποίησης πελάτη, τον αριθμό κορυφών του γραφήματος και το σύνολο ακμών του. Τα δύο τελευταία στοιχεία είναι τα ελάχιστα με τα οποία μπορεί να αναπαρασταθεί ένα γράφημα. Η μέθοδος *addObjectToCluster(String serviceName,DistribObj dobj)* προσθέτει το αντικείμενο – εξυπηρετητή *dobj* στα αντικείμενα που διαχειρίζεται το cluster και του δίνει το λογικό όνομα *serviceName*. Η μεθοδος *addObjectInClientToCluster(String serviceName,DistribObj dobj)* κάνει ακριβώς το ίδιο, μόνο που εδώ το αντικείμενο *dobj* βρίσκεται στον client host, ενώ στην προηγούμενη περίπτωση όχι. Ο λόγος για τον οποίο χρησιμοποιούνται διαφορετικές μέθοδοι στις δύο αυτές περιπτώσεις θα γίνει κατανοητός με το εξής παράδειγμα: Έστω ότι το cluster διαθέτει 3 αντικείμενα εξυπηρετητές και ότι 2 clients έχουν ταυτόχρονη πρόσβαση σε αυτό. Ο ένας χρησιμοποιείται για να εκτελέσει λειτουργίες server και ο άλλος όχι. Κατά συνέπεια ο πρώτος έχει στην διάθεση του 4 αντικείμενα – εξυπηρετητές, ενώ ο δεύτερος 3. Για να αντιμετωπιστεί αυτό επιλέχθηκε να διατηρούνται χωριστά οι servers που εκτελούνται στους client hosts από τους servers που βρίσκονται στο ίδιο τοπικό δίκτυο με το cluster. Η μέθοδος *addClient(p96020.client.Updatable client)* προσθέτει ένα πελάτη στο cluster. Μέσα στην υλοποίηση της γίνεται η ανάθεση του αριθμού ταυτοποίησης του. Τέλος η μέθοδος *setPolicy(int whichPolicy)* θέτει την πολιτική επιλογής αντικειμένου - εξυπηρετητή για το συγκεκριμένο cluster.

Τα interface που περιγράψαμε παραπάνω είναι remote, δηλαδή (ενδέχεται) τα αντικείμενα που τα υλοποιούν να βρίσκονται σε διάφορετικούς hosts. Στο σχήμα 12, περιέχονται δύο επιπλέον τα οποία δεν είναι remote. Αυτά έχουν σχέση με την παρακολούθηση και την διαχείριση του φορτίου και είναι το *LoadInfo* και το *LoadBalancer*. Βρίσκονται στον ίδιο host με το cluster του οποίου τα αντικείμενα παρακολουθούν. Το *LoadBalancer* έχει μόνο μία μέθοδο η οποία επιστρέφει το λογικό όνομα του server ο οποίος θα χρησιμοποιηθεί για την επίλυση. Το *LoadInfo* έχει ως σκοπό την παρακολούθηση του φορτίου κάθε εξυπηρετητή και την υλοποιήση των πολιτικών επιλογής τους κάτι που γίνεται με τις μεθόδους *doRoundRobin, doRandom, getShortestQueue, getLeastUtilised* για κάθε πολιτική. Επίσης η μέθοδος *updateLoad* ενημερώνει τις νέες τιμές βαθμού χρησιμοποίησης και ουράς για τον εξυπηρετήτη με το λογικό όνομα που δίνεται ως όρισμα.

Κάτι που, ενδεχομένως, πρέπει να αποσαφηνιστεί είναι η χρήση των λογικών ονομάτων. Στις περιπτώσεις του cluster αλλά και των αντικειμένων – εξυπηρετητών (όχι αυτών που κατεβαίνουν στον client) ο ρόλος τους είναι διπλός. Πρώτον είναι τα ονόματα με τα οποία δηλώνονται τα αντίστοιχα αντικείμενα στο RMI Registry (ή στην Naming Service), έτσι ώστε οι clients τους να μπορούν αρχικά να αποκτήσουν reference σε αυτά. Επίσης είναι αυτά που χρησιμοποιούνται ως κλειδιά για την αποθήκευση και ανάκτηση των references στους hashtables του cluster.

Τα παραπάνω interfaces συνθέτουν την βασική αρχιτεκτονική του συστήματος. Στην πραγματικότητα υπάρχουν πολλές ακόμα κλάσεις. Η πλήρης απεικόνιση του συστήματος φαίνεται στο παρακάτω διάγραμμα κλάσεων της UML.

Στο διάγραμμα δεν φαίνονται ούτε οι μέθοδοι ούτε οι ιδιότητες των κλάσεων. Ούτε όλες οι κλάσεις που αναπαριστώνται είναι σημαντικές Μερικές όπως για παράδειγμα αυτές που καταλήγουν σε *_Stub* είναι οι τοπικοί αντιπρόσωποι των Remote Αντικειμένων και έχουν παραχθεί από τον *RMI Compiler*. Μερικές είναι βοηθητικές κλασεις , όπως για παράδειγμα η *ServerState, η LineWeight* κτλ και θα είχαν αναπαρασταθεί ως structs αν κάτι τέτοιο υπήρχε στην Java. Δεν έχουν ουσιαστική σημασία για την λειτουργία του συστήματος Τέλος άλλες αναπαριστούν στοιχεία του user interface (όπως οι: *GraphicalViewPanel, ResultsPanel, ResTableModel, SettingsFrame*) και βέβαια υπάρχουν και οι κλάσεις που υλοποιούν τα remote interfaces που περιγράψαμε νωρίτερα. Ο κώδικας όλων των κλάσεων φαίνεται στο παράρτημα. Εδώ θα αναφέρουμε μερικά σχόλια για τις πιο σημαντικές κλάσεις του σχήματος.



Σχήμα 13. Διάγραμμα Κλάσεων

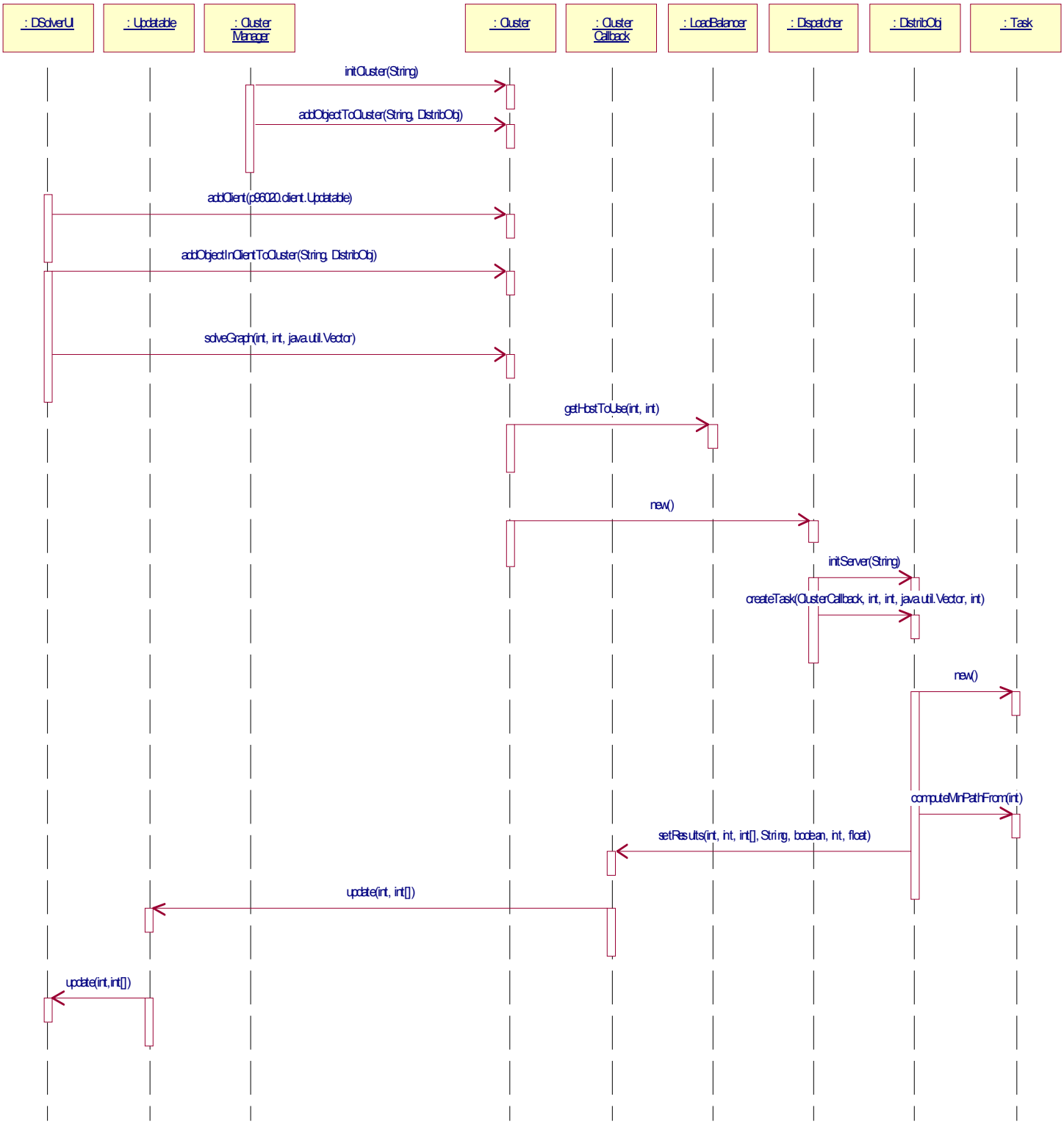
Η κλάση *Task* αναπαριστά μια 'εργασία' την οποία αναλαμβάνει να εκτελέσει το κάθε αντικείμενο - εξυπηρετητής. Στην συγκεκριμένη περίπτωση, η εργασία αναπαριστά το ελάχιστο κομμάτι του προβλήματος που μπορεί να επιλυθεί παράλληλα, δηλαδή την εύρεση του μικρότερου μονοπατιού από μια κορυφή σε κάθε άλλη του γραφήματος. Σε άλλα περιβάλλοντα μπορεί να υπάρξει προσαρμογή της λειτουργίας της συγκεκριμένης κλάσης. Για λόγους συγχρονισμού αλλά και υλοποίησης του αλγορίθμου κάθε Task θα περιλαμβάνει όλο το γράφημα. Επίσης θα περιλαμβάνει και την κορυφή από την οποία θα αναζητείται το

μικρότερο μονοπάτι. Τα δεδομένα αυτά θα περνούν στον constructor της κλάσης. Η μέθοδος *createAdjacencyMatrix()* δημιουργεί τον πίνακα γειτνίασης από το σύνολο κορυφών και ακμών του γραφήματος. Ο συγκεκριμένος πίνακας αναπαριστά εσωτερικά το γράφημα. Η μέθοδος *computeMinPathFrom()* παίρνει ως όρισμα την την κορυφή και βρίσκει μέσω του πίνακα γειτνίασης το ελάχιστο μονοπάτι. Είναι δηλαδή η μέθοδος που υλοποιεί τον αλγόριθμο του Dijkstra. Για την λειτουργία της χρησιμοποιεί την μέθοδο *findMinLabel()* η οποία ανήκει και αυτή στην ίδια κλάση. Η κλάση *Task* αρχικοποιείται και ελέγχεται από την κλάση *DSolver*. Η τελευταία υλοποιεί το interface *DistribObj*. Τα αντικείμενα Tasks είναι τοπικά με τα *DΣolvers*. Εκτός όμως από τις μεθόδους που περιγράψαμε νωρίτερα η συγκεκριμένη κλάση ορίζει και δύο επιπλέον μεθόδους, οι οποίες καλούνται από τοπικά αντικείμενα. Η πρώτη (*notify()*) καλείται από στην περίπτωση που το συγκεκριμένο αντικειμενο έχει φορτωθεί και εκτελείται στον client. Στόχος της είναι να περάσει μια reference του client στο αντικείμενο , έτσι ώστε τα αποτελέσματα του αλγορίθμου να μην στέλνονται μέσω του cluster, αλλά άμεσα. Η δεύτερη (*calculateStats()*) καλείται από ένα νήμα το οποίο σε σταθερά χρονικά διαστήματα υπολογίζει τις τιμές του φορτίου για το συγκεκριμένο αντικείμενο εξυπηρετητή. Το νήμα αυτό είναι η κλάση *Timer*.

Οι κλάσεις *DsolverApplet, DsolverApplication, ClusterManager* είναι τα σημεία έναρξης της εφαρμογής για τον client (applet και αυτόνομη εφαρμογή) και για τον server. Για τα αντικείμενα εξυπηρετητές στην υλοποίηση με RMI ενεργοποιούνται αυτόματα ενώ στην υλοποίηση για CORBA τα σημεία έναρξης είναι σα αντικείμενα *DSolver*.

Το αντικείμενο τύπου *Dispatcher* είναι υπεύθυνο για την ‘διανομή’ των αιτήσεων επίλυσης σε όλα τα αντικείμενα *DistribObj*. Η λειτουργία αυτή γίνεται ταυτόχρονα για όλα τα αντικείμενα εξυπηρετητές και για τον σκοπό αυτό το συγκεκριμένο αντικείμενο εκτελείται σε ξεχωριστό νήμα.

Το τελευταίο τμήμα του συστήματος που δεν έχει σχολιαστεί ακόμα αφορά το τμήμα του client της εφαρμογής. Συγκεκριμένα η κλάση *DSolverUI* παριστάνει το τμήμα της εφαρμογής που παρουσιάζεται στον χρήστη το οποίο φαίνεται σε προγούμενη εικόνα. Επίσης υλοποιεί και το interface *Updatable* . Άλλα συστατικά του user interface όπως ο πίνακας των αποτελεσμάτων, η περιοχή σχεδίασης του γραφήματος υλοποιούνται από άλλες κλάσεις.

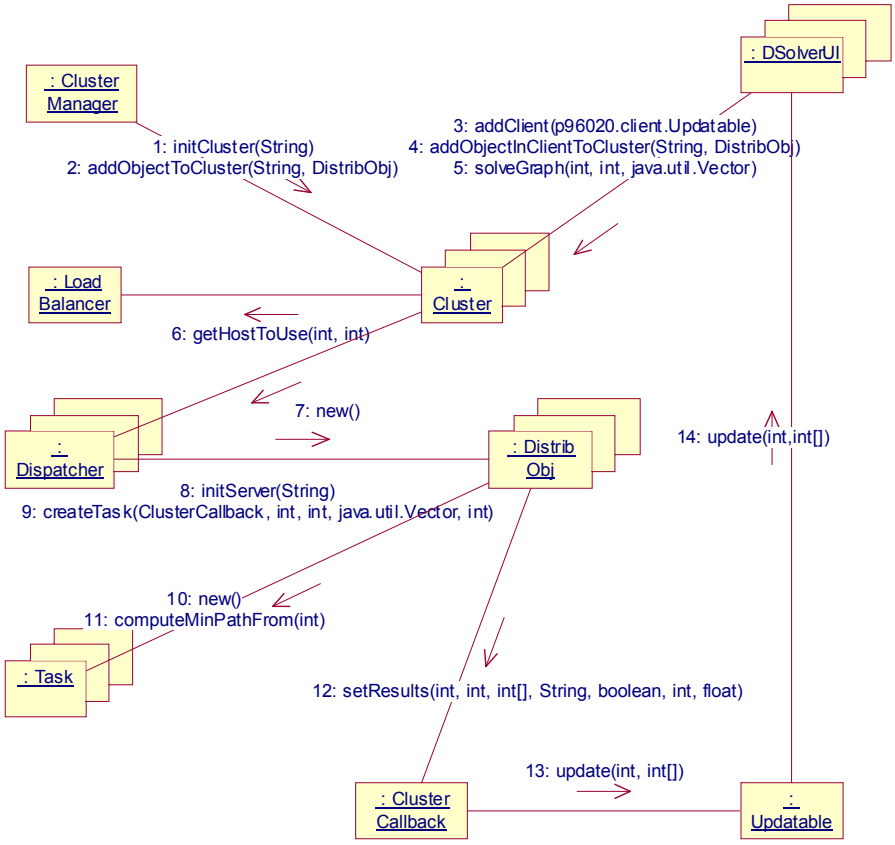


Σχήμα 14. Διάγραμμα Ακολουθίας (Sequence Diagram).

Αφού έγινε η περιγραφή των συστατικών της εφαρμογής θα περιγραφεί τώρα η αλληλεπίδραση τους. Αυτή φαίνεται στα διαγράμματα αλληλεπίδρασης της UML, δηλαδή στο διάγραμμα ακολουθίας (*Sequence Diagram*) και στο διάγραμμα συνεργασίας (*Collaboration Diagram*). Το πρώτο δίνει έμφαση στην χρονική σειρά των αλληλεπιδράσεων δύο αντικειμένων, ενώ το δεύτερο στα μηνύματα που ανταλλάσονται. Τα δύο αυτά διαγράμματα είναι πάντως σχεδόν ισοδύναμα.

Σχολιάζοντας το διάγραμμα 14, παρατηρούμε ότι αρχικά ο χρήστης ξεκινά την εφαρμογή , δηλαδή φορτώνει την κλάση ClusterManager, η οποία αρχικοποιεί το αντικείμενο Cluster και προσθέτει όλα τα αντικείμενα – εξυπηρετητές σε αυτό. Έπειτα η εφαρμογή client υπακούοντας σε μια εντολή του χρήστη επικοινωνεί με το cluster περνώντας του το γράφημα. Πρώτα έχει δηλώσει τον εαυτό της στο σύνολο των πελατών που διαχειρίζεται το cluster. Προαιρετικά προηγουμένως μπορεί να έχει ενεργοποιήσει και το αντικείμενο εξυπηρετητή που έχει κατεβεί στον host της. Το cluster τότε ενεργοποιεί τον LoadController για να βρεί σε ποιον server θα στείλει την αίτηση. Για κάθε κορυφή του γραφήματος ενεργοποιεί αντίστοιχα νήματα για να μεταβιβάσει ταυτόχρονα τις αιτήσεις. Για κάθε νέα αίτηση δημιουργείται νέο Task και υπολογίζεται το ελάχιστο μονοπάτι. Τα αποτελέσματα μεταφέρονται με χρήση των callbacks αρχικά στο cluster και μετά στον client όπου παρουσιάζονται στον χρήστη.

Το ίδιο σενάριο παρουσιάζεται και στο διάγραμμα συνεργασίας που φαίνεται αμέσως στην συνέχεια. Εδώ αξίζει να παρατηρήσουμε ότι μπορούμε να αναπαραστήσουμε το γεγονός ότι για κάθε cluster υπάρχουν πολλοί clients και αντικείμενα - εξυπηρετητές και για κάθε αντικείμενο-εξυπηρετητή πολλά tasks. Επίσης αξίζει να επισημάνουμε πως το CORBA / RMI χρησιμοποιείται σε δύο επίπεδα. Μία για την επικοινωνία και κλήση μεθόδων του cluster από τον client, και σε δεύτερο επίπεδο για την επικοινωνία και κλήση μεθόδων των αντικειμένων από το cluster.



Σχήμα 15. Διάγραμμα Συνεργασίας (Collaboration Diagram)

5. ΣΥΜΠΕΡΑΣΜΑΤΑ.

Στην συγκεκριμένη εργασία έγινε προσπάθεια να αναπτυχθεί ένα σύστημα κατανομής φορτίου για αντικειμενοστρεφή περιβάλλοντα τα οποία βασίζονται στο RMI και στο CORBA. Η βασική ιδέα για την εργασία αυτή βρίσκεται στο [1]. Στόχος ήταν η υλοποίηση μιας εφαρμογής με χρήση RMI (ή CORBA) ενσωματώνοντας όσο το δυνατόν περισσότερα στοιχεία από το [1], όπου η εφαρμογή:

- Θα λειτουργούσε σε περιβάλλον Object Web.
- Θα χρησιμοποιούσε τον client host για λειτουργίες του server.
- Θα επέτρεπε την δυναμική κατανομή του φορτίου στα διαθέσιμα αντικείμενα - εξυπηρετητές.

Σε ότι αφορά το πρώτο, πράγματι η εφαρμογή χρησιμοποιεί το περιβάλλον του Object Web. Το σύστημα έχει μοντελοποιηθεί με χρήση αντικειμένων. Αυτά που θα πρέπει να αλληλεπιδράσουν με τον χρήστη έχουν την μορφή ενός Java applet. Αλλά και στην πλευρά του server, η επίλυση του αλγορίθμου γίνεται από αντίστοιχα ειδικά αντικείμενα. Το πρωτόκολλο HTTP χρησιμοποιείται μόνο ως αρχικό μέσο για το κατέβασμα του applet, ενώ η υπόλοιπη αλληλεπίδραση γίνεται μέσω του πρωτοκόλλου JRMP ή του IIOP.

Δεύτερον επιτρέπει την εκτέλεση λειτουργιών που κανονικά θα περίμενε κανείς να εκτελούνται σε κάποιον server να ‘κατεβαίνουν’ και να εκτελούνται στους client. Αυτό επιτυγχάνεται σχετικά εύκολα χάρις το χαρακτηριστικό του κινούμενου κώδικα (mobile code) που διακρίνει την Java. Μετατρέπει δε, την τυπική client - server προσέγγιση, σε πραγματικά κατανεμημένη καθώς με την εκτέλεση λειτουργιών του client στον server έχουμε αποκέντρωση της επεξεργασίας.

Επίσης στην συγκεκριμένη εργασία γίνεται προσπάθεια για την διαχείριση του φορτίου του όλου συστήματος, δηλαδή των αιτήσεων για επίλυση ενός προβλήματος, με τρόπο ώστε όλα τα συστατικά του να συμμετέχουν εξίσου στην επεξεργασία. Η κατανομή του φορτίου γίνεται με τρόπο δυναμικό, δηλαδή κάθε αίτηση ανατίθεται σε έναν εξυπηρετητή με βάση την τρέχουσα πολιτική φορτίου. Δεν κατέστη όμως δυνατή η εξαγωγή κάποιων συμπερασμάτων για την απόδοση του συστήματος με βάση τις τιμές διαφόρων καλά ορισμένων μεγεθών, όπως για παράδειγμα ο βαθμός χρησιμοποίησης κάθε εξυπηρετητή. Σε κάθε περίπτωση πάντως η ανάθεση μέρους της επεξεργασίας στον client βοηθά την διαχείριση του περισσότερου φορτίου στην πλευρά του server.

Ένα ενδιαφέρον ερώτημα που τίθεται για την συγκεκριμένη εργασία είναι η χρησιμότητα της. Πράγματι διαβάζοντας κανείς τις παραπάνω σελίδες, ίσως αναρωτιέται αν όλα αυτά που αναφέρονται εδώ είναι εφαρμόσιμα μόνο σε καταστάσεις όπου το πρόβλημα που επιλύεται είναι ανάλογο με αυτό του αλγορίθμου του Dijkstra ή αν μπορούν να εφαρμοστούν και σε άλλες πιο πραγματικές καταστάσεις. Ενδεχομένως κάτι τέτοιο να μπορεί να γίνει. Δεν έχει

σημασία που εδώ τα αντικείμενα - εξυπηρετητές δημιουργούν Tasks τα οποία εφαρμόζουν έναν αλγόριθμο. Σε μια άλλη περίπτωση τα tasks θα μπορούσαν να έχουν πρόσβαση σε κάποια άλλα δεδομένα και να κάνουν συγκρίσεις ή αναζητήσεις σε βάσεις δεδομένων. Επίσης το cluster απλώς διαχειρίζεται DistribObjs - δεν ενδιαφέρεται για το ποια είναι η εργασία τους. Οι πελάτες απλώς ενημερώνονται για αποτελέσματα (δεν έχει σημασία που αυτά είναι αποστάσεις από κορυφές – θα μπορούσαν ίσως να είναι στατιστικά στοιχεία). Εξ'αλλου η χρήση της Αντικειμενοστρεφούς Τεχνολογίας θα μπορούσε να κάνει πιο εύκολη την μετάβαση σε ένα νέο πεδίο εφαρμογής, καθώς με επέκταση των κλάσεων cluster, distribobj κτλ. ενσωματώνονται ήδη οι τεχνικές που αναπτύχθηκαν εδώ σε αντικείμενα που ενδεχομένως να έχουν άλλες λειτουργίες. Βέβαια κάτι τέτοιο εύκολα λέγεται, πόσο εύκολα όμως υλοποιείται; Για παράδειγμα, ας υποθέσουμε ότι αντί για τον αλγόριθμο του Djikstra είχαμε ως αντικείμενο εφαρμογής της προτεινόμενης αρχιτεκτονικής ένα ηλεκτρονικό κατάστημα. Στην συγκεκριμένη περίπτωση τίθεται το θέμα της παράλληλης επεξεργασίας. Εδώ δεν θα ήταν ανάγκη τα διάφορα κατανεμημένα αντικείμενα να υλοποιούσαν ένα τμήμα της συνολικής επεξεργασίας, όπως στην περίπτωση του αλγορίθμου του Djikstra. Πιο εύκολο και ίσως πιο χρήσιμο, θα ήταν η κατανομή των αιτήσεων να ήταν το αντικείμενο της διαχείρισης του φορτίου. Θα υπήρχαν δηλαδή πανομοιότυπα αντικείμενα – εξυπηρετητές (*DistibObj*) στα οποία θα γινόταν εναλλάξ (ή με κάποιο άλλο κριτήριο), η αποστολή των αιτήσεων των πελατών από το cluster. Η προσέγγιση αυτή μοιάζει πολύ με την Round Robin DNS. Ενδεχομένως να ήταν και χρήσιμη η παραλλαγή μιας άλλης τεχνικής διαχειρίσης φορτίου από το Web (που αφορά δηλαδή ιστοσελίδες), η διαμέριση των αντικειμένων εξυπηρετητών με βάση την λειτουργία. Το cluster θα είχε δηλαδή αντικείμενα - εξυπηρετητές τα οποία θα χωρίζονταν ανάλογα με την λειτουργία τους (κάθε κατηγορία και μια *υποκλάση* της *DistribObj*). Άλλα δηλαδή ίσως ασχολούνταν με τον κατάλογο προϊόντων, άλλα με την διαχείριση των αγορών του πελάτη (καλάθι αγορών – wish list) κτλ. Βέβαια στην συγκεκριμένη περίπτωση δεν θα ήταν όλα τα αντικείμενα εξυπηρετητές κατάλληλα για εκτέλεση στον client host, όπως στην περίπτωση του Dijkstra. Δεν θα ήταν αποδοτική από πλευράς ταχύτητας αλλά και ασφάλειας, η εκτέλεση στον client host των αντικειμένων – εξυπηρετητών που έχουν πρόσβαση στην Βάση Δεδομένων του καταστήματος.

Επιπλέον επεκτάσεις της εφαρμογής αφορούν την ολοκλήρωση και την αυτοματοποίηση του συστήματος διαχείρισης φορτίου. Όπως προαναφέρθηκε, ο τρόπος διαχείρισης φορτίου στην παρούσα φάση ρυθμίζεται αποκλειστικά από την εκτέλεση της εφαρμογής κονσόλας. Μία ενδιαφέρουσα τροποποίηση θα ξεκινούσε το σύστημα με κάποια προκαθορισμένη πολιτική πχ. *Round Robin* και ανάλογα με την κατάσταση του σε δεδομένες χρονικές στιγμές θα την άλλαζε σε *Shortest Queue* ή *Least Utilized*. Επίσης στην παρούσα φάση το αντικείμενο – εξυπηρετητής, το οποίο 'κατεβαίνει' στον client host, ενεργοποιείται ουσιαστικά με την επέμβαση του χρήστη. Μία άλλη δυνατότητα θα ήταν το συγκεκριμένο

αντικείμενο να ενεργοποιούνταν μόνο από το cluster και σε περίπτωση μάλιστα που υπήρχε ανάγκη, δηλαδή υπερφόρτωση των αντικειμένων - εξυπηρετητών. Καλό θα ήταν βέβαια στην συγκεκριμένη περίπτωση να ενημερωνόταν ο χρήστης για το ότι πρόκειται να χρησιμοποιηθεί για επεξεργασία ο υπολογιστής του και το σύστημα να απαιτούσε την έγκριση του για να προχωρήσει. Τέτοιες επεκτάσεις είναι δυνατόν να υλοποιηθούν τώρα που υπάρχει, σε αρχική λειτουργία τουλάχιστον, μία υλοποίηση των όσων περιγράφονται στο [1].

Η τεχνική που έχει αναπτυχθεί εδώ είναι εφαρμόσιμη σε περιβάλλοντα Object Web, μιας αρκετά φιλόδοξης επέκτασης του Web, όπου πλέον το πρωτόκολλο HTTP θα χρησιμοποιείται μόνο για την έναρξη της εφαρμογή και η υπόλοιπη επικονωνία θα γίνεται μέσω άλλων πρωτοκόλλων όπως το IIOP, JRMP ή το πρωτόκολλο του DCOM. Το αν αυτό η επέκταση θα επικρατήσει εξαρτάται, παρά τα πλεονεκτήματα, από πολλους παράγοντες. Ο πιο σημαντικός είναι η δυνατότητα τα πρωτόκολλα αυτά να είναι ανοικτά και συμβατά . Η επιτυχία άλλωστε του HTTP οφείλεται εν μέρει στο γεγονός ότι δεν υπήρχαν ανταγωνιστές (με τόσο ευρεία αποδοχή τουλάχιστον). Ήταν ανοικτό δηλαδή με την έννοια του ότι όλοι το χρησιμοποιούσαν. Εδώ έχουμε ήδη 3 ανταγωνιστικές τεχνολογίες (CORBA, RMI, DCOM, μετρώντας μόνο τις αντικειμενοστρεφείς). Από ότι φαίνεται πάντως η τεχνολογία αυτή έχει επικρατήσει με την μία ή την άλλη μορφή μόνο στις ζώνες 2 και 3 του σχήματος 6. Στην ζώνη 1 (client) διατηρούμε το κλασικό HTML interface εμπλουτισμένο με δυναμικά στοιχεία (DHTML) κάτι που δεν φαίνεται να αλλάξει. Μπορεί να ειπωθεί δηλαδή ότι ο συνδυασμός DHTML με ASP ή CGI ή servlets για την επέκταση του web, υπερνικά τα java applets καθώς υλοποιεί καλύτερα αυτό που αναφέρεται ως thin client και από ότι φαίνεται αυτή η κατάσταση θα παραμείνει για τα επόμενα χρόνια.

ΠΑΡΑΡΤΗΜΑ

Το σύστημα έχει αναπτυχθεί σε 2 εκδόσεις: Η πρώτη χρησιμοποιεί RMI και η δεύτερη χρησιμοποιεί CORBA 2.2 . Ο ORB που χρησιμοποιείται είναι ο *Java IDL* ο οποίος έρχεται μαζί με το JDK1.2 (και με μεταγενέστερες εκδόσεις). Στην ενότητα αυτή θα παραθέσουμε τον πηγαίο κώδικα της εφαρμογής και θα τονίσουμε τα βασικά του στοιχεία. Ως σκελετός θα χρησιμοποιηθεί ο κώδικας του συστήματος σε RMI . Οι διαφορές με την έκδοση σε CORBA θα αναπτυχθούν και θα σχολιασθούν στα σημεία, όπου αυτές εμφανίζονται. Ξεκινάμε λοιπόν από τον ορισμό των *Remote Interfaces*, αυτών δηλαδή που ορίζουν τις υπηρεσίες που χρησιμοποιούν τα αντικείμενα – πελάτες και υλοποιούν τα αντικείμενα - εξυπηρετητές.

1. Remote Interfaces.

Στην παράθεση κώδικα που ακολουθεί φαίνονται τα Remote Interfaces όπως αυτά έχουν περιγραφεί με Java (RMI) και IDL (CORBA).

```
package p96020.server;

public interface Cluster extends java.rmi.Remote{

    public void initCluster(String serviceName) throws java.rmi.RemoteException;

    public void solveGraph(int clientNum,int numOfVertices,java.util.Vector edges) throws
java.rmi.RemoteException;

    public String addObjectToCluster(String serviceName,DistribObj dobj) throws
java.rmi.RemoteException;

    public String addObjectInClientToCluster(String serviceName,DistribObj dobj) throws
java.rmi.RemoteException;

    public int addClient(p96020.client.Updatable client) throws java.rmi.RemoteException;

    public void setPolicy(int whichPolicy) throws java.rmi.RemoteException;
}


public interface ClusterCallback extends java.rmi.Remote{

     public void setResults(int clientNum,int vo,int[] results,String serverName,boolean inClient,int
queueLength,float loadIndex) throws java.rmi.RemoteException;

}
```

```
package p96020.server;

public interface DistribObj extends java.rmi.Remote{

  public void initServer(String serviceName) throws java.rmi.RemoteException;

  public void createTask(ClusterCallback clustercallback,int clientNum,int
numOfVertices,java.util.Vector edges,int vo) throws java.rmi.RemoteException;

  public void setLoadIndex(float loadIndex) throws java.rmi.RemoteException;

  public void setQueueLength(int queueLength) throws java.rmi.RemoteException;

  public float getLoadIndex() throws java.rmi.RemoteException;

  public int getQueueLength() throws java.rmi.RemoteException;

  public String getServerName() throws java.rmi.RemoteException;

  public boolean isInClient() throws java.rmi.RemoteException;
}


package p96020.client;

public interface Updatable extends java.rmi.Remote{

 public void update(int vo,int[] results) throws java.rmi.RemoteException;

 public boolean isUsedForServer()  throws java.rmi.RemoteException;

 public String getServiceName()  throws java.rmi.RemoteException;

}
```

Παράθεση Κώδικα 1. Remote Interfaces σε Java

Η λειτουργίες των μεθόδων αυτών ήταν το αντικείμενο του κεφαλαίου 4 οπότε δεν θα σχολιασθούν εδώ. Παρατίθενται κυρίως για να υπάρξει κάποια σύγκριση μεταξύ του RMI και του CORBA σε ότι αφορά τις γλώσσες τις οποίες χρησιμοποιούν για να περιγράψουν τα interfaces των αντικειμένων (Java – IDL). Όπως παρατηρούμε η περιγραφή του interface σε Java είναι πολύ πιο κατανοητή από ότι σε IDL.

Συγκεκριμένα, παρατηρούμε ότι οι μέθοδοι *createTask, solveGraph* πρέπει να λάβουν ως όρισμα ένα αντικείμενο το οποίο θα τους δίνει πληροφορία για τις ακμές του γραφήματος. Αυτό το αντικείμενο στην έκδοση του RMI ορίζεται με την κλάση Edge. Στις μεθόδους που αναφέραμε περνά με την βοήθεια ενός vector (array μεταβλητού μεγέθους), του οποίου τα στοιχεία έχουν τύπο δεδομένων Edge. Στο IDL κάτι τέτοιο δεν είναι δυνατόν. Για να είχαμε σε μία μέθοδο όρισμα τύπου Edge, θα έπρεπε να ορίζαμε το interface του σε IDL. Όμως κάτι τέτοιο γίνεται για τα remote αντικείμενα και οδηγεί στην παραγωγή των stubs και skeletons. Το συγκεκριμένο αντικείμενο δεν είναι remote. Για τον σκοπό αυτό στο IDL το Edge υλοποιήθηκε ως struct, όπως φαίνεται και παρακάτω, έτσι ώστε να μπορεί να είναι όρισμα σε μέθοδο. Το παραπάνω γεγονός βέβαια δεν καθιστά το CORBA απόλυτα αντικειμενοστρεφές.

```
module p96020{
 module UtilMod{
  struct Edge{
         long start;
         long stop;
         float weight;
  };
 };
 module UpdateMod{
  interface Updatable{
    typedef sequence< long > Results;

    void update(in long vertex,in Results results);

    boolean isUsedForServer();

    string getServiceName();

  };
  interface ClusterCallback{
   typedef sequence< long > Results;

   void setResults(in long clientNum,in long vo,in Results results,in string serverName,in
boolean inClient,in long queueLength,in float loadIndex);

  };
};
 module DistribMod{
  interface DistribObj{
      typedef sequence< UtilMod::Edge > Edges;

      void initServer(in string serviceName);

      void createTask(in p96020::UpdateMod::ClusterCallback clustercallback,in long
clientNum,in long numOfVertices,in Edges e,in long vertex);

      void setLoadIndex(in float loadIndex);

      void setQueueLength(in long queueLength);

      float getLoadIndex();

      long getQueueLength();

      string getServerName();

      boolean isInClient() ;
  };
 };
 module ClusterMod{
    interface Cluster{
     typedef sequence< UtilMod::Edge > Edges;

     void initCluster(in string serviceName);

     void solveGraph(in long clientNum,in long numOfVertices,in Edges e);

     string addObjectToCluster(in string serviceName,in p96020::DistribMod::DistribObj dobj);

     string addObjectInClientToCluster(in string serviceName,in p96020::DistribMod::DistribObj
dobj);

     long addClient(in p96020::UpdateMod::Updatable client);

     void setPolicy(in long whichPolicy);
  };
 };
};
```

Παράθεση Κώδικα 2. Remote Interfaces σε IDL.

2. Το πακέτο p96020.client.

Στο συγκεκριμένο πακέτο κλάσεων του συστήματος περιέχεται η εφαρμογή με την οποία αλληλεπιδρά ο χρήστης. Το σημείο εισόδου της εφαρμογής εξαρτάται από το αν η εκτέλεση της γίνεται ως applet ή ως application. Σε κάθε περίπτωση πάντως χρησιμοποιούμε τον ίδιο κώδικα και για την έκδοση RMI και για την έκδοση CORBA.

```
package p96020.client;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DSolverApplet extends JApplet{
  public void init(){
          JPanel p=new JPanel();
          p.add(new DSolverUI(this));
          setContentPane(p);
  }
}
```

Παράθεση Κώδικα 3. Σημείο Έναρξης Applet

```
package p96020.client;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.applet.*;

public class DSolverApplication{
 public static void main(String[] args){
 if (args.length==0){
   System.out.println("***********************************************");
   System.out.println("Usage: java p96020.client.DSolverApplication <cluster_host>");
   System.exit(0);
 }
 else
 {
  new DSolverApplication(args);
 }
}
 public DSolverApplication(String[] args){
  JFrame applicationFrame=new JFrame("Dijkstra Algorithm Solver");

  applicationFrame.addWindowListener(new WindowAdapter(){
          public void windowClosing(WindowEvent e){
                                  System.exit(0);
          }
  });
  DSolverUI appUI=new DSolverUI(args);
  applicationFrame.setContentPane(appUI);
  applicationFrame.pack();
  applicationFrame.setVisible(true);
 }
}
```

Παράθεση Κώδικα 4. Σημείο Έναρξης Application

Και στις δύο περιπτώσεις πάντως δημιουργείται ένα νέο αντικείμενο της κλάσεως *DSolverUI*, οπότε από εδώ και στο εξής ο κώδικας είναι κοινός και στην περίπτωση που εκτελούμε applet και στην περίπτωση που εκτελούμε application. Η κλάση DSolverUI λοιπόν είναι αυτή που αφενός δημιουργεί το interface που βλέπει ο χρήστης και αφετέρου επικοινωνεί με το cluster το οποίο διατηρεί τον έλεγχο των αντικειμένων - εξυπηρετητών. Για την καλύτερη ΄συμπεριφορά' των συστατικών του interface της εφαρμογής κάθε διαχωρίσιμη λειτουργία εκτελείται σε ξεχωριστό νήμα. Έτσι σε διαφορετικό νήμα δημιουργείται το user interface της εφαρμογής, σε διαφορετικό νήμα δημιουργείται ένα τυχαίο γράφημα, αν επιλεγεί, και σε διαφορετικό νήμα γίνεται η επικοινωνία με το cluster. Εκτός όμως από την 'κατασκευή' του interface και την αντίδραση σε συμβάντα που παράγονται από τις αντιδράσεις του χρήστη, η συγκεκριμένη κλάση υλοποιεί και το interface *Updatable*. Αυτό σημαίνει πως μπορεί να ενημερώνεται για τα αποτέλεσματα τις λύση του αλγορίθμου. Η ενημέρωση γίνεται με χρήση της τεχνικής των callbacks, που περιγράψαμε στο κεφάλαιο 4. Η εντολή *java.rmi.server.UnicastRemoteObject.exportObject(this)* επιτρέπει την εξαγωγή του συγκεκριμένου αντικειμένου στο σύστημα RMI ώστε να μπορεί να δέχεται κλήσεις (*της* μεθόδου *update()*) από το cluster για ενημέρωση σχετικά με τα αποτελέσματα του αλγορίθμου. Στο σημείο αυτό υπάρχει μία διαφορά από την υλοποίηση σε CORBA. Συγκεκριμένα στο RMI, για να καταστεί ένα αντικείμενο remote αρκεί να υλοποιεί το(α) αντίστοιχο(α) remote interface(s). Κατά συνέπεια, ένα αντικείμενο μπορεί να υλοποιήσει όσα remote interfaces θέλει καθώς η Java δεν θέτει περιορισμούς στον αριθμό των interfaces που υλοποιεί μια κλάση. Στην περίπτωση του CORBA, όμως τα πράγματα είναι λίγο διαφορετικά. Για να καταστεί ένα αντικείμενο remote πρέπει να επεκτείνει συνήθως κάποια κλάση η οποία παράγεται από τον IDL Compiler (*_XXXImplBase ή XXXPOA*, όπου *XXX* είναι το όνομα του IDL interface). Το πρόβλημα υπάρχει αν η υλοποίηση γίνει σε Java καθώς δεν επιτρέπεται η πολλαπλή κληρονομικότητα, δηλαδή ένα αντικείμενο δεν μπορεί να επεκτείνει πάνω από μία κλάσεις. Σε περίπτωση λοιπόν που το αντικείμενο επεκτείνει ήδη κάποια κλάση (όπως εδώ που επεκτείνεται το JPanel) δεν υπάρχει δυνατότητα για να γίνει remote. Στην συγκεκριμένη περίπτωση δεν υπήρχε δυνατότητα να επεκταθεί η κλάση *_UpdatableImplBase*. Το πρόβλημα λύνεται με την τεχνική των ties, όπου στην ουσία δημιουργείται από τον idl compiler μία κλάση που επεκτείνει την ζητούμενη και όλες οι αιτήσεις μεταβιβάζονται σε αυτήν. Η μέθοδος Tie δεν χρησιμοποιείται εδώ γιατί προσθέτει σημαντική επιβάρυνση στην εφαρμογή σε ότι αφορά την ταχύτητα. Η λύση που προτιμήθηκε υλοποιεί το Updatable ως inner class της *DSolverUI* οπότε έχει πρόσβαση σε όλα τα μέλη της τελευταίας (ακόμα και στα private).

Τέλος ανάλογα με το αν έχει επιλεγεί η εκτέλεση ενός server object στον client αρχικοποιούμε ένα αντικείμενο DSolver για την το οποίο και δηλώνεται στο cluster.

```
package p96020.client;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import java.util.*;
import java.rmi.*;
import java.io.*;
import java.net.*;

import p96020.server.Cluster;
import p96020.server.Edge;


public class DSolverUI extends JPanel implements ActionListener,Updatable,Runnable{
        private JPanel inputPanel,p,settingsPanel;
        private Random randomGenerator;
        private GraphicalViewPanel graphPanel;
        private ResultsPanel resultPanel;

        private JTextField txtNumVertices,txtEdgeStart,txtEdgeStop,txtEdgeWeight;
        private JButton bGraph,bSolve,bNext,bSettings,bRandom,bConnect;
        private JTextArea status;

        private Vector edges;
        private int numOfVertices;

        private String registryHost;
        private int registryPort=1099;
        private String serviceName="Cluster";
        String originHost,codebase;

        private String clusterURL;
        private Cluster ci;

        private boolean useClient=false;
        private String myServerName="callback";

        String[] args;
        Applet applet;

        private ImageIcon graph=null,random=null,solve=null,
                clear=null,settings=null,ok=null;


        public DSolverUI(String[] args){
          originHost=args[0];
          this.args=args;
          initialize();
        }

        public DSolverUI(DSolverApplet applet){
          this.applet=applet;
          originHost=applet.getCodeBase().getHost();
          initialize();
         }


        public void initialize(){
          createUI();
          registryHost=originHost;
          edges=new Vector();
           randomGenerator=new Random();
          registryHost=originHost;

          try{
```

```
            java.rmi.server.UnicastRemoteObject.exportObject(this);
            report("Callback exported.");
           }
           catch (Exception e){
             report("Error exporting callback");
                    report(e.toString());
           }

   }

   public void run(){
           Thread current=Thread.currentThread();

           if ((current.getName()).equals("random")){
             createRandomGraph();
           }
           if ((current.getName()).equals("initGraph")){
                    initGraph();
           }
           if ((current.getName()).equals("solve")){
                    initiateSolution();
           }

         }

   public void createUI(){
     setLayout(new BorderLayout());

                    if (applet!=null)
                    {
                      graph=new ImageIcon(getURL("Classes/images/client/graph.gif"));
                      random=new ImageIcon(getURL("Classes/images/client/random.gif"));
                      solve=new ImageIcon(getURL("Classes/images/client/solve.gif"));
                      clear=new ImageIcon(getURL("Classes/images/client/clear.gif"));
                      settings=new ImageIcon(getURL("Classes/images/client/settings.gif"));
                      ok=new ImageIcon(getURL("Classes/images/client/ok.gif"));
                    }
                    else
                    {
                      graph=new ImageIcon("images/client/graph.gif");
                      random=new ImageIcon("images/client/random.gif");
                      solve=new ImageIcon("images/client/solve.gif");
                      clear=new ImageIcon("images/client/clear.gif");
                      settings=new ImageIcon("images/client/settings.gif");
                      ok=new ImageIcon("images/client/ok.gif");
                    }

                    inputPanel=new JPanel();
                    inputPanel.setBorder(BorderFactory.createLineBorder(Color.black));
                    inputPanel.setLayout(new BorderLayout());

                    JPanel p1 = new JPanel(new GridLayout(2,3));
                    p1.add(new JLabel("Enter number of vertices:"));
                    p1.setBorder(BorderFactory.createEtchedBorder());

                    txtNumVertices=new JTextField(5);
                    p1.add(txtNumVertices);

                    bGraph=new JButton("Enter Graph Info",graph);
                    bGraph.addActionListener(this);
                    bGraph.setToolTipText("Connect vertices of graph and assign weights to each
edge.");
                    p1.add(bGraph);

                    bRandom=new JButton("Random Graph",random);
                    bRandom.addActionListener(this);
                    bRandom.setToolTipText("Create random graph - very useful for large graphs");
```

```
p1.add(bRandom);

bSolve=new JButton("Solve",solve);
bSolve.addActionListener(this);
bSolve.setToolTipText("Start the solution process of the graph");
p1.add(bSolve);

bConnect=new JButton("Clear Table",clear);
bConnect.addActionListener(this);
p1.add(bConnect);

inputPanel.add(p1,BorderLayout.NORTH);
JPanel p2=new JPanel(new GridLayout(2,3));
p2.setBorder(BorderFactory.createTitledBorder("Manual Input"));

txtEdgeStart=new JTextField(5);
txtEdgeStart.setToolTipText("Enter the source of the edge.");
txtEdgeStop=new JTextField(5);
txtEdgeStop.setToolTipText("Enter the destination of the edge.");
txtEdgeWeight=new JTextField(2);
txtEdgeWeight.setToolTipText("Enter the weight of the edge.");

bNext=new JButton("Next Edge >>");
bNext.addActionListener(this);
bNext.setToolTipText("Add Edge.");

p2.add(new JLabel("Join Vertices (i,j):"));
p2.add(txtEdgeStart);
p2.add(txtEdgeStop);

p2.add(new JLabel("Enter Weight :"));
p2.add(txtEdgeWeight);
p2.add(bNext);

disableAll();
inputPanel.add(p2,BorderLayout.CENTER);

graphPanel=new GraphicalViewPanel(this);
graphPanel.setToolTipText("Displays Graphical View of The small graphs");

resultPanel=new ResultsPanel();

settingsPanel=new JPanel(new BorderLayout());
settingsPanel.setBorder(BorderFactory.createTitledBorder("Status"));
bSettings=new JButton("Settings...",settings);
bSettings.setToolTipText("Useful Configuration");
bSettings.addActionListener(this);
settingsPanel.add(bSettings,BorderLayout.SOUTH);

status = new JTextArea(7, 30);
status.setMargin(new Insets(10,10,10,10));
status.setEditable(false);
status.setToolTipText("Information Messages.");
settingsPanel.add(new JScrollPane(status));

JPanel p=new JPanel(new BorderLayout());
p.add(inputPanel,BorderLayout.NORTH);

JScrollPane jcp= resultPanel.createUI();
JSplitPane split = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
jcp , settingsPanel);
split.setOneTouchExpandable(true);
p.add(split,BorderLayout.CENTER);


JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
p , graphPanel);
```

```
                splitPane.setOneTouchExpandable(true);

               add(splitPane,BorderLayout.CENTER);
          }

          public void actionPerformed(ActionEvent e){
           Object obj=e.getSource();

           if (obj.equals(bGraph)){
             Thread t=new Thread(this,"initGraph");
                   t.start();
            }


           if (obj.equals(bSolve)){
                    Thread t=(new Thread(this,"solve"));
                    t.start();
           }

           if (obj.equals(bConnect)){
                   resultPanel.clearTable();
            }

           //next button was pressed
           if (obj.equals(bNext)){
              try{
                    int i=Integer.valueOf(txtEdgeStart.getText()).intValue();
                    int j=Integer.valueOf(txtEdgeStop.getText()).intValue();
                    int w=Integer.valueOf(txtEdgeWeight.getText()).intValue();
                    if ((i>=numOfVertices)||(j>=numOfVertices)||(i<0)||(j<0)) {
                     JOptionPane.showMessageDialog(this,"Out Of Range","Input
Error",JOptionPane.ERROR_MESSAGE);
                     txtEdgeStart.setText("");
                     txtEdgeStop.setText("");
                     return;
                    }
                    if (i==j)
                    {
                JOptionPane.showMessageDialog(this,"Loops are not allowed","Input
Error",JOptionPane.ERROR_MESSAGE);
                     txtEdgeStart.setText("");
                     txtEdgeStop.setText("");
                     return;
                    }
                    if (w<0)
                    {
                JOptionPane.showMessageDialog(this,"The weights of the edges in Dijkstra's
algorithm, must be positive integers","Input Error",JOptionPane.ERROR_MESSAGE);
                     txtEdgeWeight.setText("");
                     return;
                    }
                    int start=i;
                    int stop=j;
                    Edge edge=new Edge(start,stop,w);
                    edges.addElement(edge);
                    graphPanel.drawEdge(i,j,w);
                    txtEdgeStart.setText("");
                    txtEdgeStop.setText("");
                    txtEdgeWeight.setText("");
              }
              catch (NumberFormatException e1){
                    JOptionPane.showMessageDialog(this,"Make sure you enter numbers. Repeat
current edge.","Input Error",JOptionPane.ERROR_MESSAGE);
                    txtEdgeStart.setText("");
                    txtEdgeStop.setText("");
                    txtEdgeWeight.setText("");
                   }
```

```
    }
    if (obj.equals(bSettings)){
            new SettingsFrame();
          }
          if (obj.equals(bRandom)){
        Thread t=(new Thread(this,"random"));
                t.start();
          }
        }

        public void initGraph(){
          try{
                numOfVertices=0;
                edges.removeAllElements();
                numOfVertices=Integer.valueOf(txtNumVertices.getText()).intValue();
                enableAll();
                graphPanel.reset();
                resultPanel.clearTable();
                resultPanel.setNum(numOfVertices);
                graphPanel.drawVertices(numOfVertices);
           }
           catch (NumberFormatException e1){
                 JOptionPane.showMessageDialog(p,"Make sure you enter numbers","Input
Error",JOptionPane.ERROR_MESSAGE);
                 txtNumVertices.setText("");
               }

        }



        public synchronized void initiateSolution(){
            report("Initiating Algorithm Solution");
                disableAll();
                try{
                        clusterURL=new
String("rmi://"+registryHost+":"+registryPort+"/"+serviceName);
                        report("Looking up cluster: "+clusterURL);
         ci=(Cluster)(Naming.lookup(clusterURL));

                        if (useClient){
       report("Now adding object server in client host");
                         report("Client Host will be used for server processing");
       p96020.server.DSolver ds=new  p96020.server.DSolver(null,null);
                         ds.notify(this);
       String m=ci.addObjectInClientToCluster(getServerName()+ myServerName,ds);
       report(m);
                        }
                        else{
                         report("Not using client host for server processing");
                        }


                        //lookup cluster
                        report("Cluster found.Registering to be notified for results");
                        //get notified for results
                        int clientNum=ci.addClient(this);
                        report("My session number is : "+clientNum);
                        //start solving graph
      ci.solveGraph(clientNum,numOfVertices,edges);
                 report("Waiting for solution...");
      }
                 catch (java.rmi.NotBoundException nbe){
          JOptionPane.showMessageDialog(p,"Please Check if the cluster is exported to the
registry","Cluster Not Bound",JOptionPane.ERROR_MESSAGE);
                                report(nbe.toString());
                 }
```

```
                    catch (java.net.MalformedURLException mue){
                        report("URL Format error");
                                report(mue.toString());
                    }
                    catch (java.rmi.RemoteException re){
                            JOptionPane.showMessageDialog(p,"Please check the RMI Registry
Host And Port in settings","Cannot Find Naming Service",JOptionPane.ERROR_MESSAGE);
                        report(re.toString());
                    }
                    catch (Exception ex){
                       report("Exception " + ex.toString());
                    }
            }


            public void createRandomGraph(){
             try{
               bSolve.setEnabled(false);
               numOfVertices=0;
              edges.removeAllElements();
               numOfVertices=Integer.valueOf(txtNumVertices.getText()).intValue();
               graphPanel.reset();
               resultPanel.clearTable();
               resultPanel.setNum(numOfVertices);
               graphPanel.drawVertices(numOfVertices);int
               maxEdges=numOfVertices*(numOfVertices-1);
               int totalEdges=randomGenerator.nextInt(maxEdges);
               while (totalEdges==0){
                 totalEdges=randomGenerator.nextInt(maxEdges);
               }
                report("Total Edges : " + totalEdges);
               for (int i=0;i<(totalEdges-1);i++){
                 int v1=randomGenerator.nextInt(numOfVertices);
                 report("V1 " + v1);
                 int v2=randomGenerator.nextInt(numOfVertices);
                 report("V2 " + v2);
                  while (v1==v2)
                 {
                    v1=randomGenerator.nextInt(numOfVertices);
                     report("V1 " + v1);
                    v2=randomGenerator.nextInt(numOfVertices);
                  report("V2 " + v2);
                 }
                  int w=randomGenerator.nextInt(1000);
                 report("w " + w);
                  while (w==0){
                    w=randomGenerator.nextInt(1000);
                   }
                   int start=v1;
                   int stop=v2;
                  Edge edge=new Edge(start,stop,w);
                  edges.addElement(edge);
                  graphPanel.drawEdge(v1,v2,w);
                  bSolve.setEnabled(true);
                }
              }
               catch (NumberFormatException e1){
                        JOptionPane.showMessageDialog(p,"Make sure you enter numbers","Input
Error",JOptionPane.ERROR_MESSAGE);
                      txtNumVertices.setText("");
                     }



            }

            public void initiateConnection(){
```

```
            URL servletURL;
                    try{
          //servlet url-we always call the same servlet
                            servletURL=new
URL("http://"+originHost+"/servlet/p96020.InitServlet?message=init");
                        report("Connecting to :" + servletURL.toString()+"...");

                            //get Connection
                            URLConnection servletConnection=servletURL.openConnection();

                            //output
                            servletConnection.setDoOutput(true);
                            servletConnection.setDoInput(true);
                            servletConnection.connect();
                            report("Connected to servlet.");


        //response
        BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                        servletConnection.getInputStream()));
        String message;
                            while ((message=in.readLine()) != null){
                             report(message);
                            }
                            in.close();
                      bConnect.setEnabled(false);
                     }
                     catch(MalformedURLException e1){
                      report("Error in constructing servlet URL.");
                     }
              catch(IOException e2){
       report("Error connecting...");
                     }

          }

          //Updatable interface methods
          ////////////////////////////
          public void update(int vo,int[] results) throws java.rmi.RemoteException{
            updateTable(vo,results);
          }

    public boolean isUsedForServer()  throws java.rmi.RemoteException{
              return useClient;
    }

          public String getServiceName()  throws java.rmi.RemoteException{
            if (useClient){
                      return getServerName()+myServerName;
                    }
                    else{
                      return null;
                    }
    }
          ////////////////////////////////////
          //Updatable interface methods---end

   public void updateTable(int i,int[] s){
             resultPanel.updateTable(i,s);
   }

    public void disableAll(){
     txtEdgeStart.setEnabled(false);
     txtEdgeStop.setEnabled(false);
             txtEdgeWeight.setEnabled(false);
     bNext.setEnabled(false);
```

```
}

public void enableAll(){
 txtEdgeStart.setEnabled(true);
 txtEdgeStop.setEnabled(true);
        txtEdgeWeight.setEnabled(true);
 bNext.setEnabled(true);
}

 public void report(String text){
      String newline=System.getProperty("line.separator");
      status.append(text+newline);
      status.setCaretPosition(status.getDocument().getLength());
 }

 public String getServerName(){
           try{
                   InetAddress local=InetAddress.getLocalHost();
                   String server=local.getHostName();
                    return server+":1099/";
            }
            catch (Exception e){
              return "localhost:1099/";
            }
  }


        class SettingsFrame extends JDialog implements ActionListener{
          private JTextField txtHost,txtPort,txtName;
          private JButton bOk;
          private JCheckBox useClientHost;
          private JTextField txtMyServerName;

          SettingsFrame(){
              setLocation(100,100);
              setTitle("Settings");
              setModal(true);
              addWindowListener(new WindowAdapter(){
                        public void windowClosing(WindowEvent e){
                              dispose();
                        }
 });

 JPanel appUI=new JPanel(new BorderLayout());
 JPanel p1=new JPanel(new GridLayout(4,2));
         p1.setBorder(BorderFactory.createTitledBorder("Settings for Cluster Location"));

 JPanel p2=new JPanel(new GridLayout(2,1));
         p2.setBorder(BorderFactory.createTitledBorder("Settings for Server in Client Host"));

 p1.add(new JLabel("RMI Registry Host:"));
 txtHost=new JTextField(registryHost);
 p1.add(txtHost);
 p1.add(new JLabel("RMI Registry Port:"));
 txtPort=new JTextField((new Integer(registryPort)).toString());
 p1.add(txtPort);
 p1.add(new JLabel("Remote Object Name:"));
 txtName=new JTextField(serviceName);
 p1.add(txtName);

 p1.add(new JLabel(""));

 bOk=new JButton("OK.",ok);
 p1.add(bOk);
bOk.addActionListener(this);
```

```
    useClientHost=new JCheckBox("Use Client Host to Execute Server Functions",useClient);
    useClientHost.addActionListener(this);
    p2.add(useClientHost);

    JPanel p3=new JPanel();
    p3.add(new JLabel("Enter name for server in client host"));
    p3.add(txtMyServerName=new JTextField(12));
    txtMyServerName.setText(myServerName);
    txtMyServerName.setEnabled(useClient);
    p2.add(p3);


     appUI.add(p1,BorderLayout.CENTER);
     appUI.add(p2,BorderLayout.SOUTH);
     setContentPane(appUI);
     setResizable(false);
     pack();
     setVisible(true);
  }


  public void actionPerformed(ActionEvent e){
     Object obj=e.getSource();

     if (obj.equals(bOk)){
            registryHost=txtHost.getText();
            serviceName=txtName.getText();
            myServerName=txtMyServerName.getText();
     try{
                    registryPort=new Integer(txtPort.getText()).intValue();
     }
     catch (NumberFormatException e1){
          JOptionPane.showMessageDialog(this,"Make sure you enter numbers","Input
Error",JOptionPane.ERROR_MESSAGE);
           txtPort.setText("");
       }
                dispose();
           }

           if (obj.equals(useClientHost)){
                 useClient=useClientHost.isSelected();
                       txtMyServerName.setEnabled(useClient);
           }

  }

 }//frame

 protected URL getURL(String filename) {
      URL codeBase = applet.getDocumentBase();
      URL url = null;

      try {
         url = new URL(codeBase, filename);
      } catch (java.net.MalformedURLException e) {
         System.err.println("Couldn't create image: " +
                    "badly specified URL");
         return null;
      }

      return url;
   }


}//class
```

Παράθεση Κώδικα 5. Η κλάση DSolver

Η inner class SettingsFrame έχει ως σκοπό την εμφάνιση ενός dialog box το οποίο επιτρέπει στον χρήστη να δηλώσει αν θέλει να ενεργοποιήσει τον client host που έχει κατέβει και να πραγαμτοποιήσει άλλες τέτοιες ρυθμίσεις.

Τα μόνα συστατικά του user interface που δεν καλύπτονται από την παραπάνω κλάση είναι ο πίνακας που εμφανίζει τα αποτελέσματα καθώς επίσης και η περιοχή που σχεδιάζεται το γράφημα. Ο πίνακας δημιουργείται από την κλάση *ResultsPanel* και η περιοχή σχεδίασης του γραφήματος από την κλάση *GraphicalViewPanel*.

```
package p96020.client;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import java.util.*;


public class ResultsPanel{
  private ResTableModel resModel;
  private int numVertices=0;

  public ResultsPanel(){  }

  public void setNum(int numVertices){
   this.numVertices=numVertices;
   resModel.getAllData();
  }

  public JScrollPane createUI(){
   resModel = new ResTableModel();
   resModel.getAllData();
   JTable table = new JTable(resModel);
   table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
   table.setToolTipText("The shorthest path from vertex vi to vj");
   JscrollPane                                                                    jcp=new
JScrollPane(table,JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,JScrollPane.HORIZONTAL
_SCROLLBAR_ALWAYS);
   table.setPreferredScrollableViewportSize(new Dimension(400, 200));
   return jcp;
  }

  public void updateTable(int index,int[] s){
              resModel.updateTable(index,s);
 }

  public void clearTable(){
       resModel.getAllData();
                    resModel.fireTableRowsDeleted(0,numVertices);
  }


 class ResTableModel extends AbstractTableModel {
       static final String VERTEX = "V" ;
       Object data[][] =null ;

                    ResTableModel(){
                    }

                    public void getAllData(){
                      data=null;
```

```
                    data=new Object[numVertices][numVertices+1];
                    fireTableStructureChanged();
                  }

                  public void updateTable(int index,int[] s){
                   for (int j=0;j<s.length;j++)
                   {
                    setValueAt(new Integer(s[j]),index,j+1);
                   }

                  }

    public int getColumnCount() {
       return numVertices+1;
    }

    public int getRowCount() {
       return numVertices;
    }

    public String getColumnName(int col) {
       if (col==0)
                              return "Vertices";
                            else
                              return VERTEX + (col-1);
    }

    public Object getValueAt(int row, int col) {
         if (col==0)
                               return VERTEX + row;

                              return data[row][col];
    }

                  public void setValueAt(Object value, int row, int col) {
          if (((Integer)(value)).intValue()==(Integer.MAX_VALUE)){
                                 data[row][col] = "---";
          }
                                       else{
                                        data[row][col] = value;
                                       }
          fireTableCellUpdated(row, col);
   }
  }}
```

Παράθεση Κώδικα 6. Η κλάση ResultsPanel .

Τέλος για την κλάση σχεδίασης του γραφήματος αξίζει να τονίσουμε ότι έχει επιλεγεί οι διάφορες κορυφές να σχεδιάζονται σε κύκλο ώστε να μην έχουμε σύγχυση και επικάλυψη των διαφόρων τόξων του γραφήματος. Τέλος το πέρας του γραφήματος παριστάνεται από ένα τετράγωνο. Σε περίπτωση που το γράφημα περιέχει πάνω από 15 κορυφές δεν σχεδιάζεται τίποτα στον συγκεκριμένο χώρο.

```
package p96020.client;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import java.util.*;

public class GraphicalViewPanel extends JPanel{
  private Graphics2D g2;
```

```
 private boolean drawingVertices,tooManyVertices;
 private int totalVertices;

 Vector  lines;

 private float radius,angle,delang,dang=0.0F;
 final float TWOPI=6.283185308F;
 float pixelSize;
 final float rWidth=100.0F,rHeight=100.0F;

 int maxX,maxY,centerX,centerY;

 public GraphicalViewPanel(DSolverUI source){
   drawingVertices=false;
            tooManyVertices=false;
            lines=new Vector();
   createUI();
 }

 public void reset(){
  drawingVertices=false;
  lines.removeAllElements();
 }

 public Dimension getPreferredSize(){
       return new Dimension(500,400);
 }

 public void createUI(){
  setBorder(BorderFactory.createTitledBorder("Graph View"));
            setBackground(Color.black);
            setForeground(Color.orange);
            repaint();
 }

 void initgr(){
  Dimension d=getSize();
  maxX=d.width-20;
           maxY=d.height-20;
  pixelSize=Math.max(rWidth/maxX,rHeight/maxY);
  centerX=maxX/2;
  centerY=maxY/2;
  radius=Math.min(rWidth,rHeight)/2.0F;
}

int iX(float x){
  return Math.round(centerX+x/pixelSize);
}

int iY(float y){
  return Math.round(centerY-y/pixelSize);
}

float fx(int X){
  return (X-centerX)*pixelSize;
}

float fy(int Y){
  return (centerY-Y)*pixelSize;
}

 public void paint(Graphics g){
   initgr();
            g2=(Graphics2D)g;

   g2.setBackground(getBackground());
```

```
    g2.clearRect(0, 0, maxX+20, maxY+20);

            if (tooManyVertices)
            {
              g2.setColor(Color.orange);
              g2.setFont(new Font("ARIAL",Font.BOLD,16));
                    g2.drawString("Preview not available- Too many vertices. ",iX(0)-
(centerX/2),iY(0));
              return;
            }

    if (!drawingVertices) return;

            int size=(int)Math.min(rWidth,rHeight)/totalVertices;
            double x,y;
            delang=TWOPI/totalVertices;

            g2.setColor(Color.orange);
            for (int i=0;i<totalVertices;i++){
             angle=i*delang+dang;
             x=radius*Math.cos(angle);
             y=radius*Math.sin(angle);
             g2.setColor(Color.orange);
             g2.fill(new Ellipse2D.Double(iX((float)x), iY((float)y), size, size));
             g2.setColor(Color.green);
             g2.drawString("V"+i,iX((float)x),iY((float)y));
            }

            for (int i = 0; i < lines.size(); i++) {
             LineWeight lw=(LineWeight)lines.elementAt(i);
             g2.setColor(Color.red);
             g2.drawLine(iX(lw.currentX1),iY(lw.currentY1),iX(lw.currentX2),iY(lw.currentY2));
    g2.fillRect(iX(lw.currentX2)-2,iY(lw.currentY2)-2,4,4);
             g2.setColor(Color.yellow);
             g2.drawString(" "+Math.round(lw.weight), iX( (float) ((lw.currentX1+lw.currentX2)/2)) ,iY(
(float) ((lw.currentY1+lw.currentY2)/2) ));
   }
  }

  public void drawVertices(int n){
   drawingVertices=true;
            totalVertices=n;
            if (n>15){
              tooManyVertices=true;
            }
            else
            {
              tooManyVertices=false;
            }
   repaint();
  }

  public void drawEdge(int i, int j,float w){
   float a1=i*delang+dang;
   float a2=j*delang+dang;
            float currentX1=(float)(radius*Math.cos(a1));
   float currentY1=(float)(radius*Math.sin(a1));
   float currentX2=(float)(radius*Math.cos(a2));
   float currentY2=(float)(radius*Math.sin(a2));
            lines.addElement(new LineWeight(currentX1,currentY1,currentX2,currentY2,w));
            repaint();
  }
}
class LineWeight{
 float currentX1,currentY1,currentX2,currentY2,weight;

 LineWeight(float currentX1,float currentY1,float currentX2,float currentY2,float weight){
```

```
 this.currentX1=currentX1;
 this.currentY1=currentY1;
 this.currentX2=currentX2;
 this.currentY2=currentY2;
 this.weight=weight;
} }
```

Παράθεση Κώδικα 7. Η κλάση GraphicViewPanel.

3. Το πακέτο p96020.server

Στο συγκεκριμένο πακέτο υπάρχουν κλάσεις οι οποίες χρησιμοποιούνται είτε από τα αντικείμενα εξυπηρετητές είτε από το cluster. Ξεκινώντας θα περιγράψουμε την κλάση ClusterManager, η οποία παρουσιάζει στον χρήστη ένα user interface τύπου κονσόλας για την δημιουργία του cluster και την προσθήκη σε αυτό αντικειμένων. Η συγκεκριμένη κλάση μπορεί να θεωρηθεί ως το σημείο έναρξης της εφαρμογής από την πλευρά του server, καθώς είναι αυτή που ενεργοποιεί το cluster και εξάγει μια reference του στο σύστημα, ώστε να μπορούν να την χρησιμοποιούν οι διάφοροι clients. Επίσης επιτρέπει τον καθορισμό της πολιτικής επιλογής εξυπηρετητή για ένα συγκεκριμένο cluster. Δίνονται στον χρήστη δυνατότητες για επιλογή του επόμενου εξυπηρετητή με βάση πολιτική Round Robin, τυχαία επιλογή, επιλογή με βάση την μικρότερη ουρά ή τον μικρότερο δείκτη χρησιμοποίησης. Επίσης εμφανίζει στον χρήστη διάφορα μηνύματα για την τρέχουσα κατάσταση του συστήματος.

```
package p96020.server;

import java.rmi.Naming;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class ClusterManager implements ActionListener{
 private String registryHostForCluster="localhost";
 private int registryPortForCluster=1099;
 private String serviceNameForCluster="Cluster";

 private String registryHostForObject="localhost";
 private int registryPortForObject=1099;
 private String serviceNameForObject="DSolver";

 private Cluster ci;  //local object

 private JTextField txtHostForCluster,txtPortForCluster,txtNameForCluster;
 private JTextField txtHostForObject,txtPortForObject,txtNameForObject;
 private JTextArea status;
 private JPanel appUI;

 //Export Button
 private JButton bOkForCluster,bOkForObject;
 private JRadioButton roundRobin,random,shortestQueue,leastUtil;

 public static void main(String[] args){
  new ClusterManager();
 }

 public ClusterManager(){
```

```
  createUI();
}

public void createUI(){
   JFrame applicationFrame=new JFrame("Server Administration Console");
   applicationFrame.addWindowListener(new WindowAdapter(){
          public void windowClosing(WindowEvent e){
             System.exit(0);
          }
   });
           String sep=System.getProperty("file.separator");
           ImageIcon cl=new ImageIcon("images/server/cluster.gif");
           ImageIcon srv=new ImageIcon("images/server/server.gif");

   appUI=new JPanel(new BorderLayout());
   JPanel p=new JPanel();
   JPanel p1=new JPanel(new GridLayout(4,2));
   JPanel p2=new JPanel(new GridLayout(4,2));
   JPanel p3=new JPanel(new BorderLayout());

   p1.setBorder(BorderFactory.createTitledBorder("Cluster Information"));

   p1.add(new JLabel("Enter host for RMI Registry:"));
   txtHostForCluster=new JTextField(registryHostForCluster);
   p1.add(txtHostForCluster);
   txtHostForCluster.setEnabled(false);
   p1.add(new JLabel("Enter port for RMI Registry:"));
   txtPortForCluster=new JTextField((new Integer(registryPortForCluster)).toString());
   p1.add(txtPortForCluster);

   p1.add(new JLabel("Enter service name for cluster:"));
   txtNameForCluster=new JTextField(serviceNameForCluster);
   p1.add(txtNameForCluster);
   txtNameForCluster.addActionListener(this);

   p1.add(new JLabel(""));

   if (cl!=null) {
            bOkForCluster=new JButton("Export Cluster",cl);
   }
           else{
            bOkForCluster=new JButton("Export Cluster");
           }

   p1.add(bOkForCluster);
   bOkForCluster.addActionListener(this);

   p2.setBorder(BorderFactory.createTitledBorder("Object Server Information"));

   p2.add(new JLabel("Enter host for RMI Registry:"));
   txtHostForObject=new JTextField(registryHostForObject);
   p2.add(txtHostForObject);
   p2.add(new JLabel("Enter port for RMI Registry:"));
   txtPortForObject=new JTextField((new Integer(registryPortForObject)).toString());
   p2.add(txtPortForObject);

   p2.add(new JLabel("Enter service name for remote object:"));
   txtNameForObject=new JTextField(serviceNameForObject);
   p2.add(txtNameForObject);
   txtNameForObject.addActionListener(this);

   p2.add(new JLabel(""));

   if (srv!=null){
            bOkForObject=new JButton("Export Object",srv);
   }
          else{
```

```
            bOkForObject=new JButton("Export Object");
          }

    p2.add(bOkForObject);
    bOkForObject.addActionListener(this);

    JPanel p4=new JPanel(new GridLayout(4,1));
    p4.setBorder(BorderFactory.createTitledBorder("Server Selection"));

    roundRobin=new JRadioButton("Round Robin");
    roundRobin.setSelected(true);
    roundRobin.setToolTipText("Selects servers in a round robin fashion");
    roundRobin.addActionListener(this);
    p4.add(roundRobin);

    random=new JRadioButton("Random");
    random.addActionListener(this);
    random.setToolTipText("Selects servers randomly");
    p4.add(random);

    shortestQueue=new JRadioButton("Shortest Queue");
    shortestQueue.addActionListener(this);
    shortestQueue.setToolTipText("Selects the server with the shortest Queue");
    p4.add(shortestQueue);

    leastUtil=new JRadioButton("Least Utilized");
    leastUtil.addActionListener(this);
    leastUtil.setToolTipText("Selects the least Utilized server");
    p4.add(leastUtil);

    ButtonGroup group=new ButtonGroup();
    group.add(roundRobin);
    group.add(random);
    group.add(shortestQueue);
    group.add(leastUtil);

    p.add(p1);
    p.add(p2);

    JPanel pp=new JPanel();
    pp.setBorder(BorderFactory.createTitledBorder("Status Messages"));

    p3.add(p4,BorderLayout.WEST);
    status = new JTextArea(10, 60);
    status.setMargin(new Insets(10,10,10,10));
    status.setEditable(false);
    pp.add(new JScrollPane(status));

    p3.add(pp,BorderLayout.CENTER);

    JSplitPane split = new JSplitPane(JSplitPane.VERTICAL_SPLIT,p , p3);
    split.setOneTouchExpandable(true);


   appUI.add(split,BorderLayout.CENTER);

   applicationFrame.setContentPane(appUI);
   applicationFrame.setResizable(false);
   applicationFrame.pack();
   applicationFrame.setVisible(true);

   txtHostForObject.setEnabled(false);
   txtPortForObject.setEnabled(false);
   txtNameForObject.setEnabled(false);
   bOkForObject.setEnabled(false);
}
```

```
private void report(String text){
        String newline=System.getProperty("line.separator");
        status.append(text+newline);
        status.setCaretPosition(status.getDocument().getLength());
}


public void actionPerformed(ActionEvent e){
 Object obj=e.getSource();


 //policy selection
 if (obj.equals(roundRobin)){
   if (roundRobin.isSelected()){
                    if (ci!=null){
                                        try{
                                          ci.setPolicy(LoadController.ROUND_ROBIN);
                                        }
                                        catch (Exception e1){
                                                report(e1.toString());
                                        }
                    }
                    else{
                      JOptionPane.showMessageDialog(appUI,"Cluster not
initiated","Error",JOptionPane.ERROR_MESSAGE);
                    }
          }
 }
 if (obj.equals(random)){
    if (random.isSelected()){
                    if (ci!=null){
                      try{
                        ci.setPolicy(LoadController.RANDOM);
                      }
                    catch (Exception e1){
                              report(e1.toString());
                       }

                    }
                    else{
                      JOptionPane.showMessageDialog(appUI,"Cluster not
initiated","Error",JOptionPane.ERROR_MESSAGE);
                    }
    }
 }
 if (obj.equals(shortestQueue)){
    if (shortestQueue.isSelected()){
       if (ci!=null){
         try{
           ci.setPolicy(LoadController.SHORTEST_QUEUE);
         }
         catch (Exception e1){
            report(e1.toString());
         }

                    }
                    else{
                      JOptionPane.showMessageDialog(appUI,"Cluster not
initiated","Error",JOptionPane.ERROR_MESSAGE);
                    }
           }


 }
 if (obj.equals(leastUtil)){
    if (leastUtil.isSelected()){
                    if (ci!=null){
```

```
                        try{
                          ci.setPolicy(LoadController.LEAST_UTIL);
                        }
                        catch (Exception e1){
                                report(e1.toString());
                        }

                      }
                      else{
                        JOptionPane.showMessageDialog(appUI,"Cluster not
initiated","Error",JOptionPane.ERROR_MESSAGE);
                      }
    }
  }



  if (obj.equals(bOkForCluster)){
            registryHostForCluster=txtHostForCluster.getText();
      serviceNameForCluster=txtNameForCluster.getText();
      registryPortForCluster=(new Integer(txtPortForCluster.getText())).intValue();

            bindClusterToNS(serviceNameForCluster);

            try {
              ci.initCluster(serviceNameForCluster);
              txtHostForObject.setEnabled(true);
        txtPortForObject.setEnabled(true);
        txtNameForObject.setEnabled(true);
        bOkForObject.setEnabled(true);
            }
            catch (Exception ex){
             report(ex.toString());
             ex.printStackTrace();
            }

   }
   if (obj.equals(bOkForObject)){
            try{
             registryHostForObject=txtHostForObject.getText();
       serviceNameForObject=txtNameForObject.getText();
       registryPortForObject=(new Integer(txtPortForObject.getText())).intValue();

          String
srv="rmi://"+registryHostForObject+":"+registryPortForObject+"/"+serviceNameForObject;

                    DistribObj dobj=(DistribObj)Naming.lookup(srv);

                    if (ci!=null){
                      String message=ci.addObjectToCluster(srv,dobj);
                      report(message);
                    }
                    else{
                      JOptionPane.showMessageDialog(appUI,"Cluster not
initiated","Error",JOptionPane.ERROR_MESSAGE);
                    }

            }
            catch (Exception ex){

JOptionPane.showMessageDialog(appUI,ex.toString(),"Error",JOptionPane.ERROR_MESSAGE);
                      ex.printStackTrace();
                      report(ex.toString());
            }

   }
  }
```

```
 //exports cluster object
 private void bindClusterToNS(String serviceName){
  try{
            //Cluster will be located on the same machine with
    //cluster manager
            ci=new ClusterImplementation();
            String
serviceDetails="//"+registryHostForCluster+":"+registryPortForCluster+"/"+serviceNameForClus
ter;
            report("Creating Cluster Service " + serviceDetails + "...");
            Naming.rebind(serviceDetails,ci);
    report("Cluster Service Registered");
            report("Waiting for calls...");
  }
  catch (Exception e){

JOptionPane.showMessageDialog(appUI,e.toString(),"Error",JOptionPane.ERROR_MESSAGE);
                    report(e.toString());
  }
 }
}
```

Παράθεση Κώδικα 8. Η κλάση ClusterManager

Η επόμενη κλάση της οποίας παρατίθεται ο κώδικας αφορά την υλοποίηση του cluster. Η κλάση αυτή ονομάζεται *ClusterImplementation* και περιέχει μόνο τις υλοποιήσεις των μεθόδων που αναφέρθηκαν πιο πάνω (κεφάλαιο 4). Για τον σκοπό αυτόν εδώ παραθέτουμε μόνο τον κώδικα. Αξίζει απλώς να σημειωθεί η βοηθητική κλάση *ServerState* η οποία περιέχει τις τιμές του φορτίου για μία συγκεκριμένη reference ενός αντικειμένο - εξυπηρετητή. Στην έκδοση με RMI η συγκεκριμένη κλάση υλοποιεί και το interface ClusterCallback για να λαμβάνει ενημέρωση για την κατάσταση των αντικειμένων εξυπηρετητών. Αντίθετα στην έκδοση με CORBA το ClusterCallback υλοποιείται πάλι ως Inner class, για τους λόγους που αναφέρθηκαν και στην περίπτωση του Updatable.

```
package p96020.server;

import java.rmi.server.*;
import java.rmi.*;
import java.util.*;
import java.net.*;

public class ClusterImplementation extends java.rmi.server.UnicastRemoteObject implements
Cluster,ClusterCallback{
  LoadController loadControl;

  Hashtable availableServers;
  Hashtable clientServers;

  String callbackHost;

  String clusterURL;
  Vector allClients;

  public ClusterImplementation() throws java.rmi.RemoteException{
            super();
  }

  public void initCluster(String serviceName) throws java.rmi.RemoteException{
   try{
```

```
                    allClients=new Vector();
                    InetAddress local=InetAddress.getLocalHost();
        clusterURL="rmi://"+local.getHostName()+":"+"1099"+"/"+serviceName;
        availableServers=new Hashtable();
                    clientServers=new Hashtable();
                    loadControl=new LoadController(this);
            }
            catch (java.net.UnknownHostException e){
        clusterURL="rmi://"+"localhost"+":"+"1099"+"/"+serviceName;
        availableServers=new Hashtable();
            }
   }

  public void solveGraph(int clientNum,int numOfVertices,java.util.Vector edges) throws
java.rmi.RemoteException{
    try{
             int numVertices=numOfVertices;

     p96020.client.Updatable currClient=(p96020.client.Updatable)(allClients.elementAt(clientNum-
1));
     boolean clientUsed=currClient.isUsedForServer();

             for (int i=0;i<numVertices;i++){

               String selectedServer=loadControl.getHostToUse(i,clientNum);
               System.out.println("Min Path From Vertex "+i+" will be calculated by server :" +
selectedServer);

       ServerState state=(ServerState)(availableServers.get(selectedServer));

                     if (state == null){
                               DistribObj
currServer=((ServerState)(clientServers.get(selectedServer))).dobj;
                               System.out.println("Min Path From Vertex "+i+" will be calculated by
current server :" + currServer.getServerName());
          int vertex=i;
                        Dispatcher                                                    d=new
Dispatcher(clientNum,selectedServer,this,currServer,numOfVertices,edges,vertex);
                        d.start();
                     }
                     else{
                     DistribObj currServer=state.dobj;
                               System.out.println("Min Path From Vertex "+i+" will be calculated by
current server :" + currServer.getServerName());
          int vertex=i;
                        Dispatcher                                                    d=new
Dispatcher(clientNum,selectedServer,this,currServer,numOfVertices,edges,vertex);
                        d.start();
                     }
             }
           }
           catch (Exception e){
               e.printStackTrace();
           }
  }

  public      String      addObjectToCluster(String      serviceName,DistribObj      dobj)      throws
java.rmi.RemoteException{
    try{
                     ServerState state=new ServerState(dobj,0,0);
                     availableServers.put(serviceName,state);
               return "Object Server "+serviceName+" succesfully added to cluster. Total
servers:"+availableServers.size();;
             }
             catch (Exception e){
               e.printStackTrace();
               return e.toString();
```

```
            }
  }

  public String addObjectInClientToCluster(String serviceName,DistribObj dobj) throws
java.rmi.RemoteException{
            try{
        ServerState state=new ServerState(dobj,0,0);
                        clientServers.put(serviceName,state);
               return "Object Server In Client "+serviceName+" succesfully added to cluster.";
            }
            catch (Exception e){
               e.printStackTrace();
               return e.toString();
            }

  }

  public void setCallbackHost(String host) throws java.rmi.RemoteException{
    callbackHost=host;
  }

  public void setResults(int clientNum,int vo,int[] results,String server,boolean inClient,int
queueLength,float loadIndex) throws java.rmi.RemoteException{
    try{
            loadControl.updateLoad(server,inClient,loadIndex,queueLength);
            if ((vo!=-1)&&(results!=null)){  //not in client host
       p96020.client.Updatable c=(p96020.client.Updatable)(allClients.elementAt(clientNum-1));
              c.update(vo,results);    //view
     }
          }
          catch (Exception e){
                    e.printStackTrace();
          }
  }

  public int addClient(p96020.client.Updatable aClient){
    allClients.addElement(aClient);
          return allClients.size();
  }

  public synchronized void setPolicy(int whichPolicy){
    loadControl.updatePolicy(whichPolicy);
  }
}

class ServerState{
 int queue;
 float loadIndex;
 DistribObj dobj=null;

 ServerState(DistribObj dobj,float loadIndex,int queue){
  this.dobj=dobj;
  this.queue=queue;
  this.loadIndex=loadIndex;
 }

}
```

Παράθεση Κώδικα 9. Η κλάση ClusterImplementation

Η κατανομή των αιτήσεων για επίλυση στους εξυπηρετητές πρέπει να γίνεται ταυτόχρονα για όλους τους διαθέσιμους servers σε ένα cluster. Για τον σκοπό αυτόν υπάρχει η κλάση *Dispatcher*, η οποία εκτελείται σε ξεχωριστό νήμα για να επιτρέπει ακριβώς αυτό.

```
package p96020.server;
import java.util.Vector;
public class Dispatcher extends Thread{
  private Vector edges;
  private DistribObj d;
  private String serviceName;
  private ClusterCallback ccb;
  private int clientNum;
  private int vertex;
  private int numOfVertices;

  public Dispatcher(int clientNum,String serviceName,ClusterCallback ccb,DistribObj d,int
numOfVertices,Vector edges,int vertex){
            this.clientNum=clientNum;
            this.serviceName=serviceName;
            this.d=d;
             this.numOfVertices=numOfVertices;
             this.edges=edges;
             this.ccb=cc
b;
             this.vertex=vertex;
  }

  public void run(){
        try{
                        System.out.println("Initializing server :"+serviceName);
                        d.initServer(serviceName);
                        System.out.println("Copying Graph to server :"+serviceName);
                        d.createTask(ccb,clientNum,numOfVertices,edges,vertex);
                }
                        catch (Exception e){
                                e.printStackTrace();                    }
  }}
```

Παράθεση Κώδικα 10. Η κλάση Dispatcher

Η όλη λειτουργία της κατανομής και της διαχείρισης του φορτίου γίνεται από τον αντικείμενο LoadController, το οποίο υλοποιεί τα interfaces LoadInfo και LoadBalancer έχοντας έτσι δυνατότητα για παρακολούθηση του φορτίου του κάθε αντικειμένου Στο αντικείμενο αυτό λαμβάνονται οι αποφάσεις για το αντικείμενο το οποίο θα αναλάβει να εξυπηρετήσει την επόμενη αίτηση, ανάλογα βέβαια με την πολιτική που έχει επιλεγεί από το cluster. Η όλη διαδικασία λαμβάνει χώρα στην μέθοδο getHostToUse, η οποία ανάλογα με την πολιτική επιλέγει ποιο θα είναι το επόμενο αντικείμενο εξυπηρετητής για την επόμενη αίτηση.

```
package p96020.server;

import java.util.*;

public class LoadController implements LoadInfo,LoadBalancer{
  ClusterImplementation cluster;
  Random randomGenerator;


  public static final int ROUND_ROBIN=1;
  public static final int RANDOM=2;
  public static final int SHORTEST_QUEUE=3;
  public static final int LEAST_UTIL=4;
```

```
 private int currentPolicy;


 public LoadController(ClusterImplementation cluster){
          this.cluster=cluster;
          randomGenerator=new Random();

 }


 public void updateLoad(String serverName,boolean inClient,float util,int qLen){
  try{
   if (inClient)
          {
            String srvname=serverName;
            ServerState ss=(ServerState)(cluster.clientServers.get(srvname));
            ServerState srvstate=new ServerState(ss.dobj,util,qLen);
            cluster.clientServers.put(srvname,srvstate);
            System.out.println("Load for server " + srvname + "set to (u,q) :"+util+","+qLen);
          }
          else{
            String srvname=serverName;
            ServerState ss=(ServerState)(cluster.availableServers.get(srvname));
            ServerState srvstate=new ServerState(ss.dobj,util,qLen);
            cluster.availableServers.put(srvname,srvstate);
            System.out.println("Load for server " + srvname + "set to (u,q) :"+util+","+qLen);
          }

         }
         catch (Exception re){
          re.printStackTrace();
         }
 }

   public String getShortestQueue(int clientNum){
          System.out.println("Initializing solution for client "+clientNum+"..."+"Selected Policy is
shortest Queue");
          int min=Integer.MAX_VALUE;
          String minKey=null;

          Set keySet=cluster.availableServers.keySet();
    Object[] keySetArray=keySet.toArray();

          for (int i=0;i<keySetArray.length;i++){
       String key=(String)(keySetArray[i]);
             ServerState currState=(ServerState)(cluster.availableServers.get(key));

                  if (currState.queue<min){
                    min=currState.queue;
                          minKey=key;
                  }
          }

          p96020.client.Updatable
currClient=(p96020.client.Updatable)(cluster.allClients.elementAt(clientNum-1));
    boolean clientUsed=false;

          try{
     clientUsed=currClient.isUsedForServer();
          }
          catch (java.rmi.RemoteException e){
            e.printStackTrace();
          }

          if (clientUsed){
    String selectedServer=null;
```

```
            try{
              selectedServer=currClient.getServiceName();
            }
            catch (java.rmi.RemoteException e){
              e.printStackTrace();
            }
            ServerState currState=(ServerState)(cluster.clientServers.get(selectedServer));

            if (currState.queue<min){
                        min=currState.queue;
                    minKey=selectedServer;
            }

          }

          return minKey;

 }

   public String getLeastUtilised(int clientNum){
    System.out.println("Initializing solution for client "+clientNum+"..."+"Selected Policy is Least
Utilized");

          float min=Float.MAX_VALUE;
          String minKey=null;

          Set keySet=cluster.availableServers.keySet();
    Object[] keySetArray=keySet.toArray();

          for (int i=0;i<keySetArray.length;i++){
       String key=(String)(keySetArray[i]);
             ServerState currState=(ServerState)(cluster.availableServers.get(key));

                    if (currState.loadIndex<min){
                      min=currState.loadIndex;
                            minKey=key;
                    }
          }

          p96020.client.Updatable
currClient=(p96020.client.Updatable)(cluster.allClients.elementAt(clientNum-1));
          boolean clientUsed=false;

          try{
            clientUsed=currClient.isUsedForServer();
          }
          catch (java.rmi.RemoteException e){
            e.printStackTrace();
          }

          if (clientUsed){
            String selectedServer=null;
                    try{
         selectedServer=currClient.getServiceName();
                    }
            catch (java.rmi.RemoteException e){
              e.printStackTrace();
            }
            ServerState currState=(ServerState)(cluster.clientServers.get(selectedServer));

            if (currState.loadIndex<min){
                        min=currState.queue;
                    minKey=selectedServer;
            }

          }
```

```
            return minKey;
 }

 public float getLoadIndex(String hostName){
   return ((ServerState)(cluster.availableServers.get(hostName))).loadIndex;
 }

 //overload state occurs when the server is busy
 //for a period greater than 42secs per minute
 public String notifyOverload(){
     return null;
 }

 public String doRoundRobin(int v,int clientNum){
  try{

            String selectedServer=null;

            System.out.println("Initializing solution for client "+clientNum+"..."+"Selected Policy is
Round Robin");

            Set keySet=cluster.availableServers.keySet();
    Object[] keySetArray=keySet.toArray();

            int normalServers=keySetArray.length;
            int totalServers=normalServers;

    p96020.client.Updatable
currClient=(p96020.client.Updatable)(cluster.allClients.elementAt(clientNum-1));
            boolean clientUsed=currClient.isUsedForServer();

            if (clientUsed){
              totalServers=totalServers+1; //for policy
            }

            System.out.println("normal Servers :" + normalServers);
            System.out.println("total Servers :" + totalServers);

            int selection=v%totalServers; //here occurs the round robin

            if ((clientUsed)&&(selection==(totalServers-1))){
               selectedServer=currClient.getServiceName();
            }
            else{
                selectedServer=(String)(keySetArray[selection]);
            }

            return selectedServer;
  }
  catch (Exception e){
     e.printStackTrace();
             return null;
  }
 }

 public String doRandom(int clientNum){
    try{
      String selectedServer=null;
             System.out.println("initializing solution for client "+clientNum+"..."+"Selected Policy is
Random");


             Set keySet=cluster.availableServers.keySet();
      Object[] keySetArray=keySet.toArray();

             int normalServers=keySetArray.length;
             int totalServers=normalServers;
```

```
    p96020.client.Updatable
currClient=(p96020.client.Updatable)(cluster.allClients.elementAt(clientNum-1));
            boolean clientUsed=currClient.isUsedForServer();

            if (clientUsed){
              totalServers=totalServers+1; //for policy
            }

            System.out.println("normal Servers :" +normalServers);
            System.out.println("total Servers :" + totalServers);

            int selection=randomGenerator.nextInt(totalServers);

            if ((clientUsed)&&(selection==(totalServers-1))){
             selectedServer=currClient.getServiceName();
            }
            else{
              selectedServer=(String)(keySetArray[selection]);
            }

            return selectedServer;

          }
          catch (Exception e){
    e.printStackTrace();
            return null;
          }
}
public String getHostToUse(int currVertex,int clientNum){
  String next;
          switch (currentPolicy)
          {
            case ROUND_ROBIN:   {
                        next=doRoundRobin(currVertex,clientNum);
                                                    break;
                        }
            case RANDOM:       {
                        next=doRandom(clientNum);
                                                    break;
                        }
            case SHORTEST_QUEUE: {
                                next=getShortestQueue(clientNum);
                                                            break;
                        }
            case LEAST_UTIL :   {
                        next = getLeastUtilised(clientNum);
                                                            break;
                        }
            default:                        {
                        next=doRoundRobin(currVertex,clientNum);
                                                    break;
                        }

          }
          return next;
}
public void updatePolicy(int whichPolicy){
  currentPolicy=whichPolicy;
          switch (currentPolicy)
          {
            case ROUND_ROBIN:   {
                        System.out.println("Current Policy set to:Round Robin");
                                                    break;
                        }
            case RANDOM:       {
                        System.out.println("Current Policy set to:Random");
```

```
                        break;
                    }
            case SHORTEST_QUEUE: {
                                    System.out.println("Current Policy set to:Shortest Queue");
                        break;
                    }
            case LEAST_UTIL :   {
                        System.out.println("Current Policy set to:Least Utilized");
                                                    break;
                    }

          }
 }

}
```

Παράθεση Κώδικα 11. Η κλάση Load Balancer

Κάθε αντικείμενο που θα χρησιμοποιηθεί ως εξυπηρετητής υλοποιεί όπως είπαμε το interface DistribObj. Αυτό εδώ γίνεται από τα αντικείμενα της κλάσεως Dsolver. Στην έκδοση του RMI τα αντικείμενα αυτά ενεργοποιούνται αυτόματα από μια διεργασία την RMI Daemon, η οποία ξεκινά από τον χρήστη σε κάθε υπολογιστή που θα φιλοξενήσει αντικείμενο εξυπηρετητή. Περισσότερα για την διαδικασία ενεργοποίησης στην επόμενη παράγραφο. Η βασική μέθοδος εδώ, όπως προείπαμε είναι η createTask, η οποία δημιουργεί και ξεκινάει την επίλυση ενός task για κάθε κορυφή για την οποία καλείται. Το όρισμα clientNum που υπάρχει καθορίζει τον πελάτη, που σε συνδυασμό με το callback για το cluster προσδιορίζουν το πού θα γίνει η ενημέρωση για τα αποτελέσματα. Επίσης στην κλάση αυτή υπάρχουν και 2 μεταβλητές οι οποίες παρακολουθούν το φορτίο του κάθε DistribObj. Αυτές είναι το το μήκος της ουράς και ο βαθμός χρησιμοποίησης του κάθε object server. Υπολογίζονται σε τακτά χρονικά διαστήματα ανάλογα με την τιμή της παραμέτρου INTERVAL με την βοήθεια του νήματος Timer. Παρακάτω φαίνεται ο κώδικας και για τις δύο αυτές κλάσεις.

```
package p96020.server;

import java.util.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.net.*;
import java.io.*;

public class DSolver implements DistribObj{
        String serviceName;
        String server;

   public boolean isInClientHost=false;
        private p96020.client.DSolverUI dsui;

        //load information
        private float loadIndex;     //server utilization
        private int queueLength;
   private long startBusyTime;
        private long stopBusyTime;
        private long busyTime;
        private boolean isBusy=false;

        public final long INTERVAL=10000; //in milliseconds
```

```
        public DSolver(ActivationID id,MarshalledObject data) throws
java.rmi.RemoteException{
                        Activatable.exportObject(this,id,0);
        }


   //constructor used for client processing
        public void notify(p96020.client.DSolverUI dsui){
       this.dsui=dsui;
                isInClientHost=true;
        }


   //called by cluster
        public void initServer(String serviceName) throws java.rmi.RemoteException{
    queueLength=0;
         loadIndex=0;
         this.serviceName=serviceName;
    server=getServerName();
    Timer t=new Timer(this);
         t.start();
        }


        public void createTask(ClusterCallback ccb,int clientNum,int numOfVertices,Vector
e,int vo) throws java.rmi.RemoteException{
                System.out.println("Computing min path from vertex " +vo + " by server " +
serviceName);

                        if (queueLength==0){
                          isBusy=true;
                          startBusyTime=System.currentTimeMillis();
                        }

                        queueLength++;
          Task t=new Task(numOfVertices,e,vo);
                        int[] results=t.computeMinPathFrom(vo);
                        queueLength--;

                        if (queueLength==0){
                          isBusy=false;
                          stopBusyTime=System.currentTimeMillis();
                                busyTime=(stopBusyTime-startBusyTime);
                        }

                try{
                         if (!isInClientHost){

ccb.setResults(clientNum,vo,results,getServerName(),isInClientHost,queueLength,loadIndex);
                    }
                    else{  //server in client host
                     dsui.updateTable(vo,results);
                     System.out.println("Updated view status");
                     ccb.setResults(clientNum,-
1,null,getServerName(),isInClientHost,queueLength,loadIndex);
                    }
                  }
                  catch (Exception e1){
                    e1.printStackTrace();
                  }


        }

        public  void calculateStats(){
         try{
```

```
     loadIndex=(busyTime/INTERVAL);

              if (!isInClientHost){
                System.out.println(" Queue length for server " + getServerName() + " is:  " +
queueLength );
                System.out.println(" Server Utilization for server " + getServerName() + " is: "   +
loadIndex);
              }
              else{
        dsui.report(" Queue length for server " + getServerName() + " is:  " + queueLength );
        dsui.report(" Server Utilization for server " + getServerName() + " is: "    + loadIndex);
      }

             }
             catch (Exception e){
              System.out.println("Exception!!!");
             }
            }

            public  void setLoadIndex(float loadIndex) throws java.rmi.RemoteException{
              this.loadIndex=loadIndex;
            }

            public void setQueueLength(int queueLength) throws java.rmi.RemoteException{
     this.queueLength=queueLength;
            }

            public float getLoadIndex() throws java.rmi.RemoteException{
              return loadIndex;
            }

            public int getQueueLength() throws java.rmi.RemoteException{
      return queueLength;
            }

            public String getServerName() throws java.rmi.RemoteException{
               try{
                     InetAddress local=InetAddress.getLocalHost();
        server=local.getHostName();
        return
                      serviceName;
                     }
                     catch (Exception e){
                      return serviceName;
                     }
            }

            public boolean isInClient() throws java.rmi.RemoteException{
               return isInClientHost;
            }}
```

Παράθεση Κώδικα 12. Η κλάση Dsolver.

```
package p96020.server;

public class Timer extends Thread{
 DSolver d;

 public Timer(DSolver d){
  this.d=d;
 }

 public void run(){
   while (true){
     try{
        d.calculateStats();
                sleep(d.INTERVAL);
     }
     catch (Exception ex){
      ex.printStackTrace();
     }
   }
```

Παράθεση Κώδικα 13 Η κλάση Timer

```
package p96020.server;

import java.util.Vector;

public class Task{
 int vo;
 int numOfVertices;
 Vector edges;
 private int[][] adjacency;
 private int n;

 Task(int numOfVertices,Vector edges,int vo){
  this.vo=vo;
          this.edges=edges;
          this.numOfVertices=numOfVertices;
          createAdjacencyMatrix();
 }

 public void createAdjacencyMatrix(){
             n=numOfVertices;
                    adjacency=new int[n][n];
                    for (int i=0;i<n;i++){
                              for (int j=0;j<n;j++){
                        if (i==j) {
                                        adjacency[i][j]=0;
                        }
                                       else{
                                        adjacency[i][j]=Integer.MAX_VALUE;
                                       }
                              }
                    }

                    for (int i=0;i<edges.size();i++){
                              Edge e=(Edge)(edges.elementAt(i));
        int start=e.getStart();
        int stop=e.getStop();
                       adjacency[start][stop]=e.getWeight();
                    }

          }


          public int[] computeMinPathFrom(int vo){
```

```
            long startTime=System.currentTimeMillis();

            boolean[] s=new boolean[n]; //visited nodes
            int[] dist=new int[n];      //distances


            int index=vo;

            for (int i=0;i<n;i++){     //initialization
             s[i]=false;               //not visited
                    dist[i]=adjacency[index][i];
            }

            s[index]=true;
            dist[index]=0;

            for (int i=0;i<n-2;i++){
                    int u=findMinLabel(dist,s);

                    if (u==-1) {
                      continue;
                    }

                    s[u]=true;               //add to visited

                    for (int w=0;w<n;w++){     //for every vertex

                      if ((!s[w])&&(adjacency[u][w]<Integer.MAX_VALUE)){  //if not visited and edge
exists

                        if (dist[u]+adjacency[u][w]<dist[w]){
                          dist[w]=dist[u]+adjacency[u][w];  //there is a shortest path
                        }

                      }
                    }

            }


            long stopTime=System.currentTimeMillis();
      //busyTime=(stopTime-startTime);

      return dist;
    }



          private  int findMinLabel(int[] dist,boolean[] s){
            int min=Integer.MAX_VALUE;
            int index=-1;

            for (int i=0;i<n;i++){
       if ((dist[i]<min)&&(!s[i])){
                      min=dist[i];
                      index=i;
                    }
            }

          return index;
    }


}
```

Παράθεση Κώδικα 14. Η κλάση Task.

Η μόνη κλάση που δεν έχει περιγραφεί είναι η Edge η οποία όπως προαναφέραμε παριστάνει ένα τόξο του γραφήματος. Ο κώδικας της φαίνεται παρακάτω:

```
package p96020.server;
import java.io.*;

public class Edge implements Serializable{
 private int start;
 private int stop;
 private int weight;
 public Edge(int start,int stop,int weight){
   this.start=start;
   this.stop=stop;
   this.weight=weight;
 }
 public int getWeight(){
   return weight;
 }
 public int getStart(){
   return start;
 }
public int getStop(){
   return stop;
 }
}
```

Παράθεση Κώδικα 15. Η κλάση Edge

4. Οδηγίες Εκτέλεσης.

Στην ενότητα αυτή παρέχονται κάποιες χρήσιμες οδηγίες για τις κατάλληλες ρυθμίσεις που πρέπει να γίνουν για την εκτέλεση της εφαρμογής. Η αναφορά θα γίνει με βάση τα παραδοτέα αρχεία της δισκέττας. Συγκεκριμένα τόσο για την έκδοση του CORBA όσο και για την έκδοση RMI τα αρχεία που είναι απαραίτητα βρίσκονται στον κατάλογο *Deployment*, οργανωμένα σε 3 καταλόγους για τον client, το cluster και τα αντικείμενα εξυπηρετητές αντίστοιχα.

Συγκεκριμένα λοιπόν στην περίπτωση που η εφαρμογή εκτελείται ως applet οι ρυθμίσεις από την πλευρά του client είναι ελάχιστες. Απλώς, στην περίπτωση που δεν έχει εγκατασταθεί το Java Plug-In θα ξεκινήσει αυτόματα η διαδικασία εγκατάστασης του μόλις φορτωθεί η ιστοσελίδα.

Στην περίπτωση που η εφαρμογή client εκτελείται αυτόνομα πρέπει να εξασφαλιστεί ότι στο μηχάνημα είναι παρόν το περιβάλλον εκτέλεσης της Java (*Java 2 Runtime Environment – J2RE*) το οποίο περιέχει την Java VM και άλλα χρήσιμα προγράμματα όπως η COSNaming Service, RMI Registry κτλ. Η εφαρμογή έχει αναπτυχθεί για το *J2RE1.3*. 'Θεωρητικά' μπορεί να χρησιμοποιηθεί οποιαδήποτε έκδοση μετά την 1.2. Καλό θα ήταν επίσης οι σχετικοί κατάλογοι (bin) του περιβάλλοντος αυτού να αναφέρονται στην μεταβλητή PATH του λειτουργικού συστήματος. Και στις δύο περιπτώσεις (applet – application) οι σχετικές κλάσεις βρίσκονται στο αρχείο JAR (*Java ARchive*) client.jar. Η εκτέλεση της αυτόνομης εφαρμογής γίνεται με το batch *application.bat* (ή πληκτρολογούμε *java -jar client.jar <host>*). Κατά την εκτέλεση της εφαρμογής πρέπει να δοθεί ως όρισμα το όνομα του host, όπου βρίσκεται το cluster. Αλλιώς η εφαρμογή θα τερματίσει αμέσως.

Στην πλευρά του server τώρα έχουμε 2 περιπτώσεις: Αρχικά έχουμε τον host που φιλοξενεί το cluster και έπειτα έχουμε τους hosts στους οποίους θα εκτελούνται τα αντικείμενα εξυπηρετητές. Σε κάθε περίπτωση στα μηχανήματα που θα πρέπει να χρησιμοποιηθούν πρέπει να είναι παρών όπως και πριν το J2RE. Επίσης σε κάθε περίπτωση πρέπει να είναι εγκατεστημένες τουλάχιστον οι κλάσεις *DistribObj.class, DSolver.class, DSolver_Stub.class, ClusterCallback.class, ClusterImplementation_Stub.class, Cluster.class, Timer.class, Task.class, Edge.class, Updatable.class* σε κάποιον Web Server σε ένα γνωστό URL. Για την αποφυγή απρόβλεπτων προβλήματων καλό είναι να μπούν όλες οι κλάσεις στο url αυτό. Επίσης πρέπει εκεί να διατηρηθεί η δομή των πακέτων της Java.

Στην πρώτη περίπτωση εκτελούμε το batch *Cluster Manager.bat* (ή πληκτρολογούμε *java -jar -Djava.rmi.server.codebase=**<arg>** cluster.jar*, όπου arg το παραπάνω URL). Επίσης πρέπει να λειτουργεί η Naming Service[4] (*Naming Service.bat*). Στο σημείο αυτό πρέπει να προσέξει κανείς το port και το host name το οποίο χρησιμοποιεί για να εξάγει το cluster να

[4] Λέγοντας Naming Service εννοούμε και το RMI Registry.

ταυτίζονται με τον host και το port στο οποίο λειτουργεί η Naming Service. Επίσης είναι σημαντικό το ότι ο client πρέπει να αναζητήσει το cluster με το ίδιο όνομα που δόθηκε στην Naming Service. Επισημαίνουμε και πάλι ότι το cluster πρέπει να εκτελείται στον ίδιο host με τον Web Server φορτώνει το applet.

Στην περίπτωση των αντικειμένων εξυπηρετητών τα πράγματα είναι κάπως πιο πολύπλοκα. Τα σχετικά αρχεία βρίσκονται στον κατάλογο */Deployment/Objects*. Αρχικά μετακινούμε το αρχείο policy στον root κατάλογο (c:\) Μετά εκτελούμε την Naming Service (*Naming Service.bat*). και τον RMI Daemon (*RMI Daemon.bat*). Τέλος εκτελούμε το πρόγραμμα setup (εντολή: *java -Djava.security.policy=**<arg1>** -Djava.rmi.server.codebase=**<arg2>** Setup.jar **<arg3>***) όπου το όρισμα 1 δέχεται το path για το αρχείο policy, το όρισμα 2 δέχεται το url για τις κλάσεις και το όρισμα 3 είναι το λογικό όνομα με το οποίο θα αναφερόμαστε στον συγκεκριμένο αντικείμενο – εξυπηρετητή. Στο σημείο αυτό πρέπει να επισημάνουμε ότι το πρόγραμμα setup δεν είναι αυτό που ενεργοποιεί τα αντικείμενα. Είναι αυτό που πρακτικά λέει στον RMI Daemon **πώς** θα γίνει η ενεργοποίηση, η οποία θα συμβεί στην πραγματικότητα όταν κταφθάσει η πρώτη κλήση μεθόδου για αυτά (*lazy activation*). Κατά συνέπεια υπάρχει περίπτωση να ένα λάθος (πχ. URL) να μην γίνει αντιληπτό με την εκτέλεση του setup αλλά όταν πρέπει να ενεργοποιηθούν τα αντικείμενα.

Στην έκδοση CORBA (Java IDL) τα πράγματα είναι πιο απλά. Η δομή των καταλόγων διατηρείται ίδια. Απλώς εκκινούμε την Naming Service (*Naming Service.bat*) για το cluster και κάθε αντικείμενο εξυπηρετήτη (*java -jar objects.jar **<Naming Service Host>** **<Naming Service Port>** **<Logical Name>***). Εδώ δεν χρειάζεται η μετακίνηση των κλάσεων σε γνωστό URL.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1]. Anne Aldous: Distribution Of Objects in an Object Web, University Of Miami, May 1999.

[2]. Robert Orfali, Dan Harkey, Jeri Edwards: Client-Server Survival Guide (third edition, Wiley Computer Publishing, 1998).

[3]. Robert Orfali, Dan Harkey, Jeri Edwards: 'CORBA, Java and The Object Web', Byte 22,October 1997, 95-8.

[4]. Robert Orfali, Dan Harkey: Client – Server Programming with Java and CORBA. (second edition) Wiley Computer Publishing, 1998.

[5]. Eric Evans and Daniel Rodgers "Using Java Applets and CORBA for Multi-User Distributed Applications" , *http://www.arlut.utexas.edu/~islwww/ird/JavaCORBA.html*

[6]. E.D. Kaltz, M. Butler, R. McGrath "A scalable HTTP server : The NCSA prototype.", Computer Networks and ISDN Systems, 27, 1994.

[7]. Daniel M. Dias, William Kish, Rajat Mukherjee and Renu Tewari "A scalable and highly available Web Server".

[8]. Gunter Rackl: "Load Distribution For CORBA Environments", Technische Universitat Munchen

[9]. Scott M. Baker and Bongki Moon: "Distributed Cooperative Web Servers".

[10]. Steve Vinoski,"CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments", IEEE Communications, February 1997, 35, 2.

[11]. "Java Remote Method Invocation Specification".

[12]. Remote Method Invocation, Java 2 SDK Documentation.

[13]. Φώτω Αφράτη και Γιώργος Παπαγεωργίου, 'Αλγόριθμοι : Μέθοδοι Σχεδίασης και Ανάλυση Πολυπλοκότητας', Εκδόσεις Συμμετρία, Αθήνα 1993.

[14]. Ιωάννης Μανωλόπουλος, 'Δομές Δεδομένων' (τόμος Α), Εκδόσεις Art Of Text, Θεσσαλονίκη 1992.

[15]. Akira Andoh, Simon Nash, 'RMI over IIOP' , Javaworld ,Δεκέμβριος 1999, URL: *http://www.javaworld.com/javaworld/jw-12-1999/jw-12-iiop.html*

[16]. Bill McCarty and Luke Cassady – Dorion, 'Java Distributed Objects', Sams Publishing, 1999.

[17]. Joel M.Crichlow , 'The Essence of Distributed Systems', Prentice Hall, 2000.

[18]. Maqelang Institute, 'Fundamentals of RMI' URL: *http://java.sun.com/*.

[19]. Jon Siegel , 'A Preview Of CORBA 3', Internet Watch , May 1999.