ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΣΥΣΤΗΜΑΤΩΝ ΜΕΤΑΔΟΣΗΣ ΠΛΗΡΟΦΟΡΙΑΣ & ΤΕΧΝΟΛΟΓΙΑΣ ΥΛΙΚΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΕΥΦΥΩΝ ΕΠΙΚΟΙΝΩΝΙΩΝ & ΔΙΚΤΥΩΝ ΕΥΡΕΙΑΣ ΖΩΝΗΣ

Σχεδίαση και Ανάπτυξη Συστήματος Ανακάλυψης και Επικοινωνίας Υπηρεσιών πάνω στην Πλατφόρμα OSGi

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Δημήτρη Η. Τσάμη

Επιβλέπων: Ιάκωβος Στ. Βενιέρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2006

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΣΥΣΤΗΜΑΤΩΝ ΜΕΤΑΔΟΣΗΣ ΠΛΗΡΟΦΟΡΙΑΣ & ΤΕΧΝΟΛΟΓΙΑΣ ΥΛΙΚΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΕΥΦΥΩΝ ΕΠΙΚΟΙΝΩΝΙΩΝ & ΔΙΚΤΥΩΝ ΕΥΡΕΙΑΣ ΖΩΝΗΣ

Σχεδίαση και Ανάπτυξη Συστήματος Ανακάλυψης και Επικοινωνίας Υπηρεσιών πάνω στην Πλατφόρμα OSGi

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Δημήτρη Η. Τσάμη

Επιβλέπων: Ιάκωβος Στ. Βενιέρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14^{η} Ιουλίου 2006.

.....	.....	.....
Ι. Βενιέρης	Γ. Στασινόπουλος	Δ. Κακλαμάνη
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν. Καθηγήτρια Ε.Μ.Π.

Αθήνα, (Ιούλιος 2006).

.....
Δημήτρης Η. Τσάμης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Οι σύγχρονοι υπολογιστές είναι σχεδόν στην εντέλεια τους δικτυωμένοι. Αρχικά, η κύρια χρήση του διαδικτύου ήταν η ανάκτηση πληροφοριών, αλλά όλο και περισσότερο στρέφεται στην χρήση υπηρεσιών που προσφέρουν απομακρυσμένοι κόμβοι. Σε αυτό έχει βοηθήσει και το SOAP, που είναι ένα πρωτόκολλο για απομακρυσμένη κλήση υπηρεσιών στο οποίο όλοι φαίνονται να έχουν συμφωνήσει. Επίσης, σχετικά πρωτόκολλα επιτρέπουν την ανακάλυψη υπηρεσιών με βάση τον τύπο τους, ώστε να μην χρειάζεται να θυμάται ο χρήστης την διεύθυνση της υπηρεσίας. Ένα από αυτά είναι και το SLP.

Σκοπός της διπλωματικής αυτής εργασίας είναι η σχεδίαση και ανάπτυξη συστήματος ανακάλυψης και επικοινωνίας υπηρεσιών σε περιβάλλον τοπικού δικτύου. Χρησιμοποιήσαμε τα υπάρχοντα πρωτόκολλα SOAP και SLP πάνω από την πλατφόρμα OSGi, που μας προσφέρει την δυνατότητα δυναμικής εγκατάστασης, εκκίνησης και σταματήματος υπηρεσιών. Το OSGi είναι μια service-oriented πλατφόρμα που διευκολύνει τη χρήση υπηρεσιών που είναι διαθέσιμες τοπικά. Επεκτείναμε την πλατφόρμα έτσι ώστε να είναι δυνατή η χρήση απομακρυσμένων υπηρεσιών, με τρόπο παρόμοιο αυτού των τοπικών υπηρεσιών.

Λέξεις-κλειδιά: OSGi, SLP, service discovery, SOAP, web services, publish-subscribe, SOA, component-based architectures

Abstract

Modern computers seem to be networked almost entirely. Originally, the internet was used for information retrieval, but this paradigm has been rapidly changing towards remote service usage. The development of SOAP, a remote procedure call protocol, has greatly helped in that direction with its widespread acceptance. Moreover, other protocols allow service discovery by service type, eliminating the need to remember the address of a service. SLP is such a protocol.

The goal of this diploma thesis was to design and implement a system that offers service discovery and communication in a local network environment. The existing protocols SOAP and SLP have been used, on top of the OSGi platform, which offers dynamic installation, starting and stopping of services. OSGi is a service-oriented and component-based platform that greatly simplifies the usage of local services. We extended the platform to allow for remote service usage, while retaining an interface similar to that used for local services.

Key words: OSGi, SLP, service discovery, SOAP, web services, publish-subscribe, SOA, component-based architectures

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Σκοπός της διπλωματικής εργασίας ήταν η σχεδίαση και η υλοποίηση ενός συστήματος ανακάλυψης κι επικοινωνίας υπηρεσιών σε περιβάλλον τοπικού δικτύου. Επίσης, ανάμεσα στους σκοπούς ήταν και η χρήση ενός δυναμικού περιβάλλοντος, όπου οι υπηρεσίες μπορούν να εκκινηθούν και να σταματηθούν δυναμικά (χωρίς να επηρεάζεται η λειτουργία του συνολικού συστήματος). Τέλος, θα πρέπει να είναι εύκολο να επεκτείνουμε την πλατφόρμα μας με την προσθήκη νέων υπηρεσιών.

Προφανώς αναγνωρίζουμε ότι για τα επιμέρους προβλήματα έχουν προταθεί πολλές λύσεις. Δεν ήταν στα σχέδια μας να υλοποιήσουμε κάτι που ήδη υπάρχει. Στόχος μας ήταν η μελέτη των ιδιοτήτων των διαφόρων λύσεων, η επιλογή των προτιμότερων και ο συνδυασμός τους σε ένα ολοκληρωμένο πλαίσιο που θα είναι χρήσιμο και εύχρηστο.

Η πρώτη σημαντική απόφαση που πάρθηκε ήταν η χρήση της πλατφόρμας OSGi. Το OSGi παρέχει ένα component-based και service-oriented περιβάλλον. Βασίζεται στη γλώσσα Java κι επιτρέπει τη δυναμική εγκατάσταση, εκκίνηση και σταμάτημα πακέτων λογισμικού. Το κάθε πακέτο (bundle) παρέχει κάποιες υπηρεσίες που μπορούν να χρησιμοποιήσουν τα υπόλοιπα πακέτα. Προωθείται ο διαχωρισμός διαπροσωπείου και υλοποίησης, αφού η κάθε δηλωμένη υπηρεσία πρέπει να υλοποιεί κάποιο διαπροσωπείο. Πέρα από το όνομά τους, οι υπηρεσίες χαρακτηρίζονται και από τα properties, που είναι ζεύγη "key=value". Είναι δυνατόν πολλά αντικείμενα να υλοποιούν την ίδια υπηρεσία και το καθένα να έχει τα δικά του properties. Ωστόσο, το OSGi υποστηρίζει τη χρήση μόνο υπηρεσιών που είναι διαθέσιμες τοπικά κι ακριβώς αυτή τη συμπεριφορά επεκτείναμε.

Για την ανακάλυψη των υπηρεσιών χρησιμοποιήσαμε το πρωτόκολλο SLP. Μελετήσαμε κι άλλα πρωτόκολλα, αλλά αυτά προσέφεραν επιπλέον λειτουργίες πέραν της ανακάλυψης. Θέλοντας να αποφύγουμε την περιττή πολυπλοκότητα επιλέξαμε τελικά το SLP, που είναι κι ανοικτό πρότυπο. Επίσης, επιθυμούσαμε τη λειτουργία χωρίς την ανάγκη ύπαρξης κάποιου κεντρικού εξυπηρετητή και το SLP υποστηρίζει πλήρως αυτό τον τρόπο λειτουργίας. Ένα ακόμα θετικό χαρακτηριστικό του SLP είναι ότι οι υποστηρίζεται η αναζήτηση με βάση τα attributes των υπηρεσιών, κάτι αντίστοιχο με τα properties του OSGi. Η αναζήτηση στο SLP γίνεται στέλνοντας mul-

ticast μηνύματα που περιέχουν τον τύπο της υπηρεσίας και τα επιθυμητά attributes. Όσοι λαμβάνουν τα μηνύματα ελέγχουν αν εξυπηρετούν κάποια υπηρεσία που να ικανοποιεί τα κριτήρια και σε αυτή την περίπτωση απαντάνε με unicast μήνυμα που περιέχει τη διεύθυνση της υπηρεσίας.

Για την απομακρυσμένη κλήση των υπηρεσιών χρησιμοποιήθηκε το SOAP. Το SOAP έχει επικρατήσει στο χώρο των web services. Τα μηνύματα του κωδικοποιούνται στη γλώσσα XML και στέλνονται πάνω από HTTP. Αυτό έχει ως αποτέλεσμα τη δυνατότητα χρήσης του οπουδήποτε, αφού το HTTP είναι καθολικά αποδεκτό. Το SOAP είναι ανεξάρτητο από γλώσσες και πλατφορμές. Με το SOAP προσδιορίζουμε ποια μέθοδος θέλουμε να κληθεί, μεταφέρουμε τις παραμέτρους στο απομακρυσμένο μηχάνημα, γίνονται οι όποιοι υπολογισμοί και τελίκα μας επιστρέφεται το αποτέλεσμα. Όμως, δεν μπορεί να μεταφερθεί οποιοδήποτε αντικείμενο πάνω από το διαδίκτυο. Το SOAP προβλέπει την κωδικοποίηση για μερικούς βασικούς τύπους δεδομένων και για τους υπόλοιπους πρέπει να προσδιορίσουμε εμείς την αναπαράστασή τους γράφοντας έναν serializer.

Πιστεύουμε ότι οι τεχνολογίες που χρησιμοποιήσαμε ανταποκρίνονται στις σύγχρονες ανάγκες. Όλο και περισσότερο ο κόσμος του λογισμικού στρέφεται προς την παροχή δικτυακών υπηρεσιών (αρκεί να αναφέρουμε το παράδειγμα του Google). Επίσης, η ζήτηση για εύκολη κι ευφυή διαχείριση δικτυακών μηχανών οδηγεί στη χρήση μεθόδων αυτόματης ανακάλυψης υπηρεσιών, καταργώντας την ανάγκη στατικής ρύθμισης με διευθυνσεις υπηρεσιών. Τέλος, όσον αφορά το OSGi η SOA (Service Oriented Architecture) γίνεται όλο και πιο δημοφιλής, ενώ οι απαιτήσεις για υψηλή διαθεσιμότητα επιβάλλουν τη χρήση τεχνολογιών lifecycle management.

Η δομή του κειμένου που ακολουθεί έχει ως εξής: στα Κεφάλαια 2 έως 5 παρουσιάζονται οι τεχνολογίες που αναφέρθηκαν και το Κεφάλαιο 6 είναι αφιερωμένο στην υλοποίηση της εργασίας μας. Πιο συγκεκριμένα, στο Κεφάλαιο 2 εξετάζεται το SLP, καθώς κι άλλες τεχνολογίες ανακάλυψης υπηρεσιών. Στο Κεφάλαιο 3 παρουσιάζεται η πλατφόρμα OSGi κι αναλύονται οι ευκολίες που παρέχει. Το Κεφάλαιο 4 επικεντρώνεται στις απομακρυσμένες κλήσεις υπηρεσιών, με σαφή έμφαση στο SOAP. Στο Κεφάλαιο 5 περιγράφεται πώς συνδυάσαμε όλες τις προηγούμενες τεχνολογίες προς την επίτευξη των σκοπών μας. Τέλος, στο Κεφάλαιο 6 αναφέρονται ιδέες για μελλοντική επέκταση της εργασίας μας.

Κεφάλαιο 2

Service Location Protocol (SLP)

2.1 Εισαγωγή

Στο παρελθόν για να χρησιμοποιήσει μια υπηρεσία ο χρήστης θα έπρεπε να ξέρει ένα αναγνωριστικό που περιγράφει πού βρίσκεται η υπηρεσία. Αυτό δημιουργούσε μεγάλο φόρτο στους διαχειριστές των δικτύων που πολλές φορές έπρεπε να ρυθμίζουν στατικά τις διάφορες εφαρμογές να χρησιμοποιούν συγκεκριμένες υπηρεσίες. Προφανώς έτσι μειώνεται πολύ η ευελιξία αφού για παράδειγμα δεν υπάρχει κάποιος έξυπνος τρόπος να ενημερωθούν οι κόμβοι για την ύπαρξη μιας νέας υπηρεσίας. Επίσης, ένας νέος κόμβος στο δίκτυο δεν μπορεί να ανακαλύψει ποιες υπηρεσίες είναι διαθέσιμες. Σε ένα κόσμο όπου οι περισσότεροι υπολογιστές τείνουν να γίνουν φορητοί είναι απαραίτητο να υπάρχει ένας μηχανισμός δυναμικής ρύθμισης των εφαρμογών που χρησιμοποιούν σε δικτυακές υπηρεσίες.

Το πρωτόκολλο Service Location Protocol (SLP) προσφέρει στους χρήστες ένα πλαίσιο μέσω του οποίου έχουν πρόσβαση σε πληροφορίες σχετικές με την ύπαρξη, την τοποθεσία και τις ρυθμίσεις δικτυακών υπηρεσιών. Ο χρήστης δεν χρείαζεται πια να γνωρίζει εκ των προτέρων το όνομα του κόμβου που φιλοξενεί την υπηρεσία. Αντίθετα, ο χρήστης ζητάει το είδος της υπηρεσίας που τον ενδιαφέρει και ένα σύνολο επιθυμητών χαρακτηριστικών. Με βάση αυτές τις πληροφορίες το SLP απαντάει με τις διευθύνσεις των κόμβων που παρέχουν τέτοιες υπηρεσίες.

Το SLP σχεδιάστηκε για λειτουργία σε τοπικά δικτύα κι όχι ως μια παγκόσμια λύση που θα καλύπτει όλο το διαδίκτυο. Χρησιμοποιείται σε τοπικά δίκτυα για την διευκόλυνση της εύρεσης εκτυπωτών, μηχανημάτων fax, εξυπηρετητών αρχείων, ταχυδρομείου ή web, ημερολόγια, κλπ.

Το SLP σχεδιάστηκε από το IETF ServLoc Working Group. Η πρώτη του έκδοση ορίζεται από το RFC 2165 [1] που κυκλοφόρησε τον Ιούνιο του 1997. Τον Ιούνιο του 1999 βγήκε και δεύτερη έκδοση, που ορίζεται από το RFC 2608 [4] κι από τότε η πρώτη έκδοση θεωρείται απαρχαιωμένη. Ως πρότυπο του IETF το SLP είναι ανοικτό πρότυπο, δηλαδή ο καθένας μπορεί να το υλοποιήσει και να το χρησιμοποιήσει, χωρίς να χρείαζεται να πληρώσει δικαιώματα.

2.2 Εναλλακτικές προσεγγίσεις

Εκτός από το SLP έχουν προταθεί κι άλλα πρωτόκολλα για τον ίδιο σκοπό, που προφανώς έχουν κάποιες διαφορές, αλλά μοιράζονται και κάποια κοινά στοιχεία. Τα παρουσιάζουμε εδώ επιγραμματικά:

2.2.1 Zeroconf

Ένα πρωτόκολλο το οποίο έχει προσελκύσει πολλή δημοσιότητα τον τελευταίο καιρό είναι το Zeroconf, γνωστό και ως Rendezvous ή Bonjour, που είναι ονόματα που έχει δώσει η Apple σε υλοποιήσεις του στους υπολογιστές Mac. Αναπτύσσεται από IETF Zeroconf Working Group, που ξεκίνησε τις εργασίες του το Σεπτέμβριο του 1999. Προς το παρόν έχουν βγει 3 draft RFCs καθώς και το RFC 3927 [11]. Κατά μία έννοια το Zeroconf είναι η εξέλιξη του παλιότερου Appletalk με το οποίο επικοινωνούσαν στο παρελθόν οι υπολογιστές Mac. Το Zeroconf είναι πολύ ευρύτερο από το SLP και καλύπτει τους εξής τομείς:

- Την ανάθεση IP διευθύνσεων στους διάφορους κόμβους του δικτύου χωρίς την ανάγκη ύπαρξης ενός εξυπηρετητή DHCP.
- Τη αντιστοίχιση ονομάτων κόμβων σε IP διευθύνσεις χωρίς την ανάγκη ύπαρξης ενός εξυπηρετητή DNS.
- Την ανεύρεση υπηρεσιών χωρίς την ανάγκη ύπαρξης κεντρικού εξυπηρετητή που να καταχωρεί όλες τις υπηρεσίες.
- Την απόδοση διευθύνσεων IP Multicast χωρίς την ανάγκη ύπαρξης εξυπηρετητή MADCAP (πρόκειται για μελλοντική δουλειά).

Είναι εμφανές πως το Zeroconf είναι ιδιαίτερα βολικό για μικρές επιχειρήσεις, αφού καταργεί την ανάγκη ύπαρξης και ρύθμισης πολλών εξυπηρετητών. Οι τεχνολογίες που έχουν προταθεί υπό την αιγίδα της IETF για την αντιμετώπιση των 3 πρώτων προβλήματων είναι οι IPv4 Link-Local Addressing[1], Multicast-DNS και DNS Service Discovery.

Παρόλο που το Zeroconf φαίνεται να είναι πιο δημοφιλές, τελικά το απορρίψαμε γιατί αφενός η προτυποποίησή του δεν έχει τελειώσει κι αφετέρου προσφέρει πολλές περισσότερες λειτουργίες από όσες μας είναι απαραίτητες.

2.2.2 Universal Plug-n-Play (UPnP)

Το UPnP (Universal Plug-n-Play αναπτύχθηκε αρχικά από την Microsoft το 1999 και τώρα αναπτύσσεται από το UPnP forum στο οποίο συμμετέχουν περίπου 750 εταιρίες. Το UPnP, όπως και το Zeroconf, είναι πολύ ευρύτερο από το SLP και καλύπτει τους παρακάτω τομείς:

[1] Κάτι που υπάρχει ήδη στο IPv6 [3]

- Αυτόματη ανάθεση διεύθυνσης IP.
- Εύκολη ανακάλυψη υπηρεσιών.
- Σύστημα ειδοποίησης για γεγονότα (event notification).
- Περιγραφή των υπηρεσιών και των συσκευών.
- Σύστημα παρουσίασης των δυνατοτήτων των συσκευών σε σελίδες html

Είναι σημαντικό να σημειώσουμε ότι το UPnP δεν ασχολείται μόνο με υπολογιστές, αλλά προσπαθεί να καλύψει κάθε είδους συσκευή που αλληλεπιδρά με άλλες συσκευές. Για αυτό κι ένα μεγάλο μέρος του UPnP αφορά τη διαχείριση των συσκευών και την παρουσίαση των δυνατοτήτων τους. Επίσης, το UPnP παρέχει κι αυτό τη δυνατότητα αυτόματης ανάθεσης διευθύνσεων IP, όπως και το Zeroconf.

Το κομμάτι του UPnP που ασχολείται με την ανακάλυψη υπηρεσιών ονομάζεται Simple Service Discovery Protocol (SSDP). Σε μια προσπάθεια να χρησιμοποιηθούν υπάρχοντα πρωτόκολλα, το SSDP χρησιμοποιεί τη δομή του HTTP, αλλά πάνω από πακέτα UDP (HTTPU) ή multicast UDP (HTTPMU). Όταν μια νέα υπηρεσία εμφανίζεται σε ένα δίκτυο πρέπει να διαφημίσει την ύπαρξή της με ένα multicast μήνυμα. Οι υπόλοιποι κόμβοι μπορούν να αποθηκεύσουν τις πληροφορίες αυτές, όπως και ο directory server που μπορεί να υπάρχει. Όταν ένας κόμβος θέλει να ανακαλύψει μια υπηρεσία μπορεί είτε να επικοινωνήσει με τον κόμβο που έστειλε την διαφήμιση ή να στείλει ένα multicast μήνυμα αναζήτησης.

Όπως γράψαμε και παραπάνω, το UPnP προσφέρει πολλά περισσότερα από όσα χρειαζόμαστε, που σημαίνει ότι αν το χρησιμοποιούσαμε θα προσθέταμε περιττή πολυπλοκότητα. Άλλωστε δεν μπορούμε να απομονώσουμε το SSDP και να το χρησιμοποιήσουμε μόνο του. Το SSDP έχει το θετικό ότι υποστηρίζει τόσο αναζητήσεις όσο και ανακοινώσεις υπηρεσιών, μια προσέγγιση που πιθανώς μειώνει τον συνολικό αριθμό μηνυμάτων που ανταλλάσσονται. Όμως, παρόλο που το UPnP έχει σύστημα περιγραφής των υπηρεσιών, αυτό δεν μπορεί να χρησιμοποιηθεί κατά την αναζήτηση έτσι ώστε να μειωθεί ο αριθμός των απαντήσεων που δεν ενδιαφέρουν τον χρήστη.

2.3 Βασικές οντότητες στο SLP

Στο SLP υπάρχουν 3 βασικές οντότητες:

User Agent Μια διεργασία που λειτουργεί για λογαριασμό του χρήστη και ανακαλύπτει υπηρεσίες. Μαζεύει πληροφορίες από τους Service Agents ή το Directory Agent.

Service Agent Μια διεργασία που λειτουργεί για λογαριασμό μιας ή περισσοτέρων υπηρεσιών και διαφημίζει την ύπαρξή τους.

Directory Agent Μια διεργασία που συλλέγει διαφημίσεις υπηρεσιών. Μπορεί να υπάρχει το πολύ ένας Directory Agent ανά κόμβο.

Από δω και πέρα θα χρησιμοποιούμε τις συντομογραφίες UA, SA και DA.

Η ύπαρξη DA στο δίκτυο είναι προαιρετική. Σε μικρές εγκαταστάσεις με μερικές δεκάδες κόμβους ένας DA ίσως δεν χρειάζεται. Σε εγκαταστάσεις όμως με χιλιάδες κόμβους η ύπαρξη DA κρίνεται απαραίτητη, για να μην υπάρχει ο κίνδυνος να κορεστεί το δίκτυο από την ανταλλαγή μηνυμάτων SLP. Δηλαδή, η υπάρξη του DA αφορά την κλιμάκωση του πρωτοκόλλου σε εγκαταστάσεις με μεγάλο αριθμό κόμβων.

2.4 Σύντομη περιγραφή της λειτουργίας

Η λειτουργία του SLP είναι διαφορετική σύμφωνα με το αν υπάρχουν DAs ή όχι. Εξετάζουμε την κάθε περίπτωση χωριστά.

Αν δεν υπάρχει DA, τότε ένας UA που θέλει να ανακαλύψει μια υπηρεσία στέλνει ένα multicast μήνυμα με την περιγραφή της υπηρεσίας. Όσοι SAs εξυπηρετούν υπηρεσίες που ταιριάζουν στην περιγραφή απαντούν με ένα unicast μήνυμα στον UA.

Στην περίπτωση που υπάρχει έστω κι ένας DA τότε η επικοινωνία γίνεται σημείο προς σημείο και ο UA στέλνει unicast μήνυμα στον DA ζητώντας πληροφορίες για την υπηρεσία που τον ενδιαφέρει. Ο DA ελέγχει τις υπηρεσίες που είναι καταγεγραμμένες σε αυτόν και απαντάει κι αυτός με unicast μήνυμα. Εξυπακούεται ότι όλοι οι SAs καταγράφουν (register) τις υπηρεσίες που εξυπηρετούν με τον αντίστοιχο DA.

Για να περιγράψουμε με μεγαλύτερη ακρίβεια το πρωτόκολλο πρέπει να εισάγουμε μερικές καινούργιες έννοιες (attributes, scopes), καθώς και να περιγράψουμε τα ήδη των μηνυμάτων που υπάρχουν.

2.5 Επιπλέον έννοιες

2.5.1 Service URL

Όπως αναφέραμε, ένας από τους σκοπούς του SLP είναι η αναζήτηση υπηρεσιών με βάση το είδος της. Για την περιγραφή των υπηρεσιών επιλέχθηκε να χρησιμοποιηθεί ένα ειδικό είδος URL, το service: URL [5]. Η προσέγγιση αυτή έχει τα εξής πλεονεκτήματα:

- Χρησιμοποιεί ένα υπάρχον και καθιερωμένο πρότυπο.
- Βασίζεται σε ήδη υπάρχουσες λύσεις για να αποφύγει τα προβλήματα με διαφορετικά σύνολα χαρακτήρων.
- Είναι συμβατή με πρωτόκολλα διαφορετικά του IP (όπως IPv6, IPX, κλπ).
- Οι browsers μπορούν ήδη να χειρίζονται URLs.

Ένα Service URL έχει την εξής δομή: "service:"<srvtype>"://"<addrspec> Δηλαδή, ξεκινάει πάντα με service: κι ακολουθεί το srvtype που περιγράφει το είδος της υπηρεσίας. Το <srvtype> (δηλαδή 'τύπος υπηρεσίας') αναλύεται σε <abstract-type> και <concrete-type> (δηλαδή αφηρημένος και συγκεκριμένος τύπος). Αυτό γίνεται για να μπορούμε να προσδιορίσουμε τον τύπο μιας υπηρεσίας με μεγαλύτερη ακρίβεια. Πάντως, το <concrete-type> δεν είναι υποχρεωτικό κι αν παραλειφθεί αναφερόμαστε σε όλες τις υπηρεσίες που ταιριάζουν με το <abstract-type>.

Το πρώτο μέρος του Service URL προσδιορίζει το είδος της υπηρεσίας, ενώ το <addrspec> προσδιορίζει που μπορεί να βρεθεί η υπηρεσία. Στην πιο συνηθισμένη του μορφή πρόκειται για μια IP διεύθυνση (ή domain name) ακολουθούμενη από την θύρα TCP. Όμως, όπως αναφέραμε αντί για διεύθυνση IP μπορεί να έχουμε για παράδειγμα μια IPX διεύθυνση.

Μερικά παραδείγματα Service URL:

service:tftp://host.example.org:333

service:printer:lpr://192.168.16.7/printer

Το δεύτερο παράδειγμα παρατηρούμε ότι προφανώς αναφέρεται σε εκτυπωτή. Για να επικοινωνήσουμε μαζί του πρέπει να χρησιμοποιήσουμε το πρωτόκολλο lpr. Επίσης, παρατηρούμε ότι δεν προσδιορίζεται η θύρα TCP, που σημαίνει ότι πρέπει να χρησιμοποιήσουμε την καθιερωμένη θύρα που αντιστοιχεί στο πρωτόκολλο lpr.

2.5.2 Attributes

Πολλές φορές ο τύπος της υπηρεσίας δεν αρκεί για να την περιγράψει πλήρως. Για το λόγο αυτό το SLP εισάγει την έννοια των attributes (χαρακτηριστικά). Πρόκειται για ζεύγη 'ιδιότητα=τιμή' που περιγράφουν τις ιδιότητες μιας υπηρεσίας. Για παράδειγμα, τα attributes ενός εκτυπωτή μπορεί να περιγράφουν αν είναι έγχρωμος, σε ποιες αναλύσεις μπορεί να τυπώσει, το όνομά του, το μοντέλο του, την τοποθεσία του κλπ. Οι πληροφορίες αυτές δίνουν τη δυνατότητα στο χρήστη να μαζέψει ικανές πληροφορίες για την υπηρεσία που πρόκειται να χρησιμοποιήσει και να κάνει πιο συγκεκριμένες αναζητήσεις.

2.5.3 Scopes

Τα scopes (εμβέλειες) είναι άλλη μια έννοια μαζί με τους DAs που εισήχθησαν για να επιτρέπουν την κλιμάκωση του πρωτοκόλλου. Ο διαχειριστής του δικτύου μπορεί να ομαδοποιήσει τις υπηρεσίες σε διαφορετικά scopes με βάση με διάφορα κριτήρια. Για παράδειγμα, μπορεί υπηρεσίες που προορίζονται για διαφορετικές ομάδες χρηστών να ανήκουν σε χωριστά scopes ή μπορεί το συνολικό δίκτυο να χωρίζεται σε επιμέρους υποδίκτυα και το καθένα να έχει το δικό του scope. Όταν ένας UA κάνει αναζήτηση υπηρεσίας και ταυτόχρονα υποδεικνύει συγκεκριμένα scopes, τότε ένας SA ή DA θα απαντήσει μόνο αν η σχετική υπηρεσία ανήκει στο ίδιο scope.

Είδος Μηνύματος	Σύντμηση	Function-ID
Service Request	SrvRqst	1
Service Reply	SrvRply	2
Service Registration	SrvReg	3
Service Deregister	SrvDeReg	4
Service Acknowledge	SrvAck	5
Attribute Request	AttrRqst	6
Attribute Reply	AttrRply	7
DA Advertisement	DAAdvert	8
Service Type Request	SrvTypeRqst	9
Service Type Reply	SrvTypeRply	10
SA Advertisement	SAAdvert	11

Πίνακας 2.1: Είδη Μηνυμάτων στο SLP

Αυτό έχει ως αποτέλεσμα να μειώνεται ο συνολικός αριθμός των μηνυμάτων που ανταλλάσσονται, κατά συνέπια μπορούν να υπάρξουν περισσότεροι κόμβοι στο δίκτυο. Φυσικά, η χρήση των scopes δεν είναι υποχρεωτική. Αν ένα μήνυμα δεν αναφέρει ρητά κάποιο scope τότε θεωρείται ότι ανήκει στο default scope.

2.6 Είδη μηνυμάτων SLP

Το RFC 2608 ορίζει διαφορετικά είδη μηνυμάτων, καθένα από τα οποία έχει ως αναγνωριστικό ένα διαφορετικό Function-ID. Στον Πίνακα 2.6 φαίνονται τα υποχρεωτικά μηνύματα.

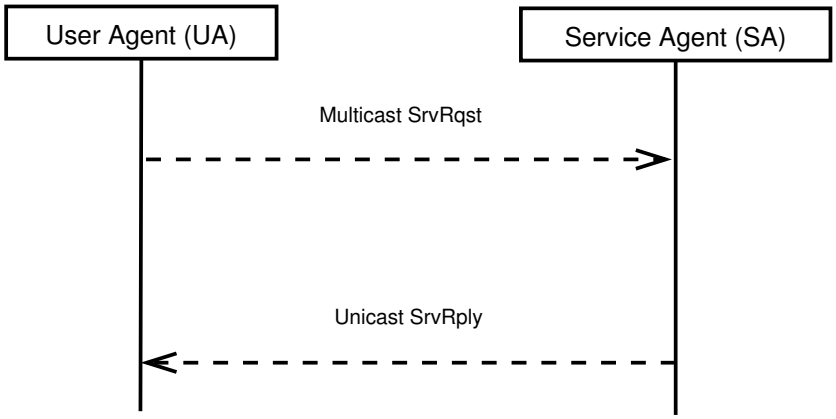
Οι UAs και οι SAs πρέπει να είναι σε θέση να χειρίζονται τα μηνύματα SrvRqst, SrvRply και DAAdvert. Επίσης, οι SAs οφείλουν να υποστηρίζουν τα μηνύματα SrvReg, SAAdvert και SrvAck. Για τους UAs και τους SAs η υποστήριξη των υπόλοιπων μηνυμάτων είναι προαιρετική.

Ο ακριβής ρόλος των μηνυμάτων θα φανεί παρακάτω όπου αναλύουμε τη λειτουργία του SLP.

2.7 Λειτουργία του SLP

Όταν ένας χρήστης αναζητά μια υπηρεσία, τότε ο UA εκπέμπει ένα μήνυμα SrvRqst. Αυτό το μήνυμα περιέχει το service type της υπηρεσίας, καθώς και τυχόν attributes και scopes. Τα attributes δίνονται σαν φίλτρο αναζήτησης LDAPv3 [2]. Για παράδειγμα, ένα φίλτρο αναζήτησης μπορεί να είναι:

```
(|(x=33)(y=foo))
```



Σχήμα 2.1: Λειτουργία του SLP χωρίς DA

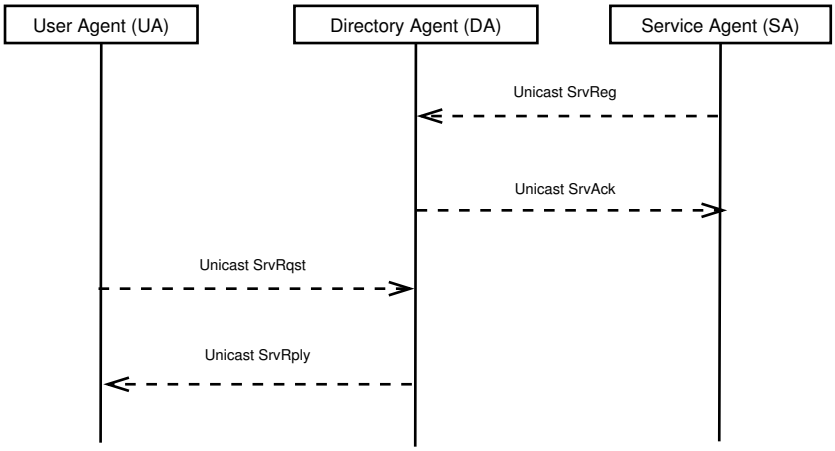
που σημαίνει ότι ζητάμε είτε το x να έχει την τιμή 33 ή το y να έχει την τιμή foo.

Ο UA μπορεί να στείλει μήνυμα SrvRqst απευθίας στους SAs κάνοντας multicast ή μπορεί να στείλει unicast μήνυμα σε έναν (ή περισσότερους) DA. Σε κάθε περίπτωση οι παραλήπτες ελέγχουν το service type της υπηρεσίας και τα πιθανά attributes και scopes. Αν το μήνυμα SrvRqst περιέχει πολλάπλα scopes τότε αρκεί ένα και μόνο scope να συμφωνεί με αυτό μιας γνωστής υπηρεσίας. Ένας SA ή DA που γνωρίζει για μια υπηρεσία που είναι συμβατή με όλα τα παραπάνω απαντάει με unicast μήνυμα SrvRply στον αρχικό UA. Το μήνυμα SrvRply περιέχει ένα ή περισσότερα Service URLs.

Αν στο δίκτυο υπάρχει κάποιος DA τότε οι SAs πρέπει να καταχωρήσουν τις υπηρεσίες που εξυπηρετούν με τον DA. Αυτό επιτυγχάνεται στέλνοντας μήνυμα SrvReg στον DA. Αυτό το μήνυμα περιέχει το Service URL της υπηρεσίας, καθώς και τα attributes και scopes που την αφορούν. Το μήνυμα SrvReg περιέχει επίσης το πεδίο Lifetime (Διάρκεια Ζωής) που καθορίζει για πόση ώρα μπορεί να αποθηκεύσει ο DA την υπηρεσία. Μετά το πέρας αυτού του χρονικού διαστήματος η υπηρεσία διαγράφεται αυτόματα από τον DA. Είναι υποχρέωση του SA να ανανεώσει την καταχώρηση της υπηρεσίας. Αν ο SA θελήσει να διαγράψει την υπηρεσία πριν εκπνεύσει η καταχώρησή της οφείλει να στείλει μήνυμα SrvDeReg στον DA.

Η λειτουργία του SLP όπως περιγράφτηκε παραπάνω, χωρίς ή με παρουσία DA φαίνεται στα σχήματα 2.1 και 2.2 αντίστοιχα.

Το πρωτόκολλο σχεδιάστηκε έτσι γιατί αποσκοπεί κατεξοχήν στο να εξυπηρετεί δυναμικά περιβάλλοντα, όπου υπηρεσίες εμφανίζονται κι εξαφανίζονται διαρκώς και με μεγάλη συχνότητα. Σε τέτοια περιβάλλοντα δεν έχει νόημα ένας DA να κρατάει τα στοιχεία μιας υπηρεσίας για πάντα, αφού δεν είναι σίγουρο ότι αυτή θα συνεχίσει να υπάρχει. Το πεδίο ¨Διάρκεια Ζωής¨ που δίνει ο SA κατά την καταχώρηση της υπηρεσίας αποτελεί μια εκτίμηση για το πόση ώρα αναμένει ο SA να παραμείνει διαθέσιμη η υπηρεσία. Αν για κάποιο λόγο η υπηρεσία χαθεί πριν από αυτό το



Σχήμα 2.2: Λειτουργία του SLP με DA

διάστημα, τότε ο SA οφείλει να ενημερώσει τον DA, αλλιώς ένας UA που θα ρωτήσει τον DA για αυτή την υπηρεσία θα πάρει μια αναφορά σε υπηρεσία που δεν υπάρχει.

Επιπλέον, ένας SA μπορεί να μεταβάλλει τα attributes μιας υπηρεσίας που έχει ήδη καταχωρήσει (incremental service registration). Αυτό επιτυγχάνεται με το να στείλει ένα SrvReg με τα καινούργια attributes κι όλα τα άλλα στοιχεία ίδια, ώστε να ανανεωθούν τα attributes που έχουν καταχωρηθεί μαζί με την υπηρεσία. Η υποστήριξη αυτής της λειτουργίας είναι προαιρετική.

Σε ένα μήνυμα SrvReg ή SrvDeReg ο DA οφείλει να απαντήσει με μήνυμα SrvAck για να ενημερώσει τον SA για την επιτυχή καταχώρηση της υπηρεσίας. Αν ο SA δεν λάβει το SrvAck σε εύλογο χρονικό διάστημα (CONFIG_RETRY δευτερόλεπτα) πρέπει να ξαναστείλει το αρχικό μήνυμα.

Με τα μηνύματα DAAdvert ένας DA διαφημίζει την ύπαρξη του και δίνει κάποια στοιχεία για τη λειτουργία του. Το πεδίο DA Stateless Boot Timestamp σημείωνει τη στιγμή που επανεκκινήθηκε για τελευταία φορά ο DA. Οι παραλήπτες του μηνύματος πρέπει να θεωρήσουν ότι όσες καταχωρήσεις είχαν γίνει πριν από αυτή τη στιγμή έχουν πια χαθεί. Επίσης, το μήνυμα DAAdvert περιέχει και τα attributes του DA, κάποια από τα οποία έχουν ειδική σημασία. Για παράδειγμα, το attribute "min-refresh-interval" ορίζει τον ελάχιστο χρόνο μεταξύ ανανεώσεων των υπηρεσιών. Αυτό συμβαίνει επειδή μπορεί κάποιοι SAs να δίνουν πολύ μικρό Lifetime οπότε μετά να χρειάζεται να ανανεώνουν συνέχεια την καταχώρηση. Αν ο DA ενοχλείται από αυτή τη συμπεριφορά δύναται να ορίσει ένα ελάχιστο χρόνο μεταξύ των ανανεώσεων και οι SAs πρέπει να σεβαστούν αυτό τον χρόνο.

Τα μηνύματα AttrRqst και AttrRply είναι προαιρετικά. Με αυτά, ένας UA μπορεί να βρει τα attributes μιας συγκεκριμένης υπηρεσίας δίνοντας το Service URL που της αντιστοιχεί. Με το μήνυμα AttrRqst ζητούνται τα attributes και με το μήνυμα AttrRply δίνονται τα στοιχεία. Η δυνατότητα αυτή προβλέπεται επειδή τα μηνύματα

SrvRply περιέχουν μόνο το Service URL κι όχι τα attributes. Ο UA μπορεί να ορίσει να επιστραφούν συγκεκριμένα μόνο attributes. Επίσης, είναι δυνατό το μήνυμα AttrRqst να περιέχει ένα service type αντί για service url και σε αυτή την περίπτωση επιστρέφονται τα attributes όλων των υπηρεσιών αυτού του τύπου.

2.8 Αυτόματη ρύθμιση των κόμβων

Όπως είδαμε μέχρι τώρα, ένας UA μπορεί να δουλέψει είτε με ή χωρίς DA. Ένας UA που πρωτοεμφανίζεται σε ένα δίκτυο πρέπει να γνωρίζει αν υπάρχει κάποιος DA· όμοια κι ένας SA πρέπει να ξέρει αν οφείλει να καταχωρήσει τις υπηρεσίες του με κάποιον DA.

Για να μπορούν οι UAs και SAs να ανακαλύψουν τον τυχόν DA το SLP προτείνει 4 τρόπους.

1. Οι UAs και SAs ρυθμίζονται στατικά.
2. Γίνεται multicast SrvRqst για να ανακαλυφθούν οι τυχόν DAs.
3. Οι τυχόν DAs ανακαλύπτονται μέσω DHCP.
4. Ο κάθε DA εκπέμπει περιοδικά multicast μηνύματα DAAdvert.

Προφανώς ο πρώτος τρόπος αποφεύγεται, γιατί η χρήση του πρωτοκόλλου γίνεται ακριβώς για να αποφευχθεί η ανάγκη στατικών ρυθμίσεων. Ο τρίτος τρόπος περιγράφεται στο RFC 2610 [6], και χρησιμοποιεί το DHCP option 78. Έχει το μειονέκτημα ότι βασίζεται σε τρίτο πρωτόκολλο, αλλά στα περισσότερα τοπικά δίκτυα ήδη υπάρχει κάποιος DHCP server (αν και δεν είναι καθόλου σίγουρο ότι θα υποστηρίζει αυτή τη λειτουργία). Το θετικό του είναι ότι ανταλλάσσονται μόνο unicast μηνύματα, σε αντίθεση με τον δεύτερο τρόπο που περιλαμβάνει και multicast μηνύματα. Όμως με τον δεύτερο τρόπο χρησιμοποιούνται μόνο SLP μηνύματα. Σε αυτή την περίπτωση το πρωτόκολλο λειτουργεί όπως όταν δεν υπάρχει DA: ο UA στέλνει ένα multicast μήνυμα SrvRqst ζητώντας το service type "service:directory-agent". Όποιος DA δει αυτό το μήνυμα απαντάει όχι με SrvRply, αλλά με DAAdvert οπότε ο UA συλλέγει όλες τις απαραίτητες πληροφορίες για τον DA.

2.9 Στοιχεία Χαμηλοτέρων Επιπέδων

Τα μηνύματα SLP μεταδίδονται πάνω από UDP, για τον εμφανή λόγο ότι το TCP δεν μπορεί να χρησιμοποιηθεί για multicast μηνύματα. Άλλωστε το TCP θα απαιτούσε να μεταδίδονται πολλά περισσότερα πακέτα. Σε περίπτωση όμως που ένας UA λάβει απάντηση που δεν χωρούσε σε ένα πακέτο UDP, τότε μπορεί να πάρει την πρωτοβουλία να συνδεθεί μέσω TCP στον αντίστοιχο SA ή DA και να ξαναζητήσει την απάντηση. Γενικά όμως είναι επιθυμητό να αποφεύγεται αυτή η συμπεριφορά.

Για το SLP έχει ήδη ανατεθεί η θύρα 427. Για multicast SrvRqst μηνύματα πρέπει να χρησιμοποιείται η διεύθυνση 239.255.255.253. Σε απομονωμένα δίκτυα μπορεί να χρησιμοποιείται broadcast αντί για multicast.

2.10 Ασφάλεια στο SLP

Το SLP σχεδιάστηκε για να προσφέρει πληροφορίες στους χρήστες του. Οι προβλέψεις που έγιναν για την ασφάλεια στο SLP δεν αφορούν τον περιορισμό της πρόσβασης στην πληροφορία, αλλά την πιστοποίηση της πηγής της πληροφορίας. Δηλαδή σκοπός είναι οι UAs να μπορούν να ελέγξουν αν οι πληροφορίες που λαμβάνουν είναι από αξιόπιστες πηγές.

Το SLP χρησιμοποιεί ψηφιακές υπογραφές και κρυπτογραφία δημοσίου κλειδιού. Μπορούν να χρησιμοποιηθούν διάφορα κρυπτογραφικα πρωτόκολλα, αλλά η υποστήριξη του DSA με SHA-1 [12] είναι υποχρεωτική. Όταν ένας UA στέλνει ένα μήνυμα SrvRqst μπορεί να περιλάβει ένα SLP SPI (SLP Security Parameter Index). Το SPI περιγράφει τα κρυπτογραφικά πρωτόκολλα που είναι αποδεκτά από τον UA. Αυτός που θα απαντήσει θα πρέπει να έχει την κατάλληλη υπογραφή για την υπηρεσία και μάλιστα σε μορφή που να συμφωνεί με το SPI. Η τυχούσα υπογραφή περιέχεται στο κομμάτι Authentication Block του μηνύματος SrvRply. Αν αυτός που απαντάει είναι ένας SA τότε παράγει μόνος του την υπογραφή για την υπηρεσία που εξυπηρετεί. Για να απαντήσει ένας DA θα πρέπει νωρίτερα όταν ο SA καταχώρησε την υπηρεσία να περιέλαβε την υπογραφή στο μήνυμα SrvReg. Επίσης, οι DAs υπογράφουν κι αυτοί τα DAAdverts κι ένας SA θα πρέπει να περιλαμβανεί υπογραφές για τις καταχωρήσεις του μόνο προς DAs που έχουν υπογράψει τα στοιχεία τους.

Παρόλες τις προβλέψεις του πρωτοκόλλου στην πράξη η δυνατότητα υπογραφής των υπηρεσιών σπάνια χρησιμοποιείται. Για να χρησιμοποιηθεί χρειάζεται πολλή προετοιμασία από την πλευρά του διαχειριστή του δικτύου, αφού πρέπει να δημιουργηθούν και να κατανεμηθούν όλα τα κλειδιά και τα πιστοποιητικά (certificates) που απαιτούνται.

2.11 Επεκτάσεις του SLP

Κατά καιρούς έχουν προταθεί διάφορες επεκτάσεις του SLP για να καλύψουν διαφορετικές ανάγκες. Πολλές φορές αυτές οι επεκτάσεις δεν είχαν ιδιαίτερη τύχη, αλλά αξίζει να αναφέρουμε μερικές.

2.11.1 Wide Area SLP (WASLP)

Όπως ήδη αναφέραμε, το SLP προορίζεται για χρήση σε τοπικά δικτύα. Αυτό συμβαίνει γιατί δεν μπορούμε multicast μηνύματα σε επίπεδο διαδικτύου χωρίς τεράστιο κόστος τόσο σε καθυστέρηση όσο και σε συμφόρηση. Μια προσπάθεια να

επεκταθεί το SLP σε μεγάλα δίκτυα ήταν το Wide Area SLP (WASLP), που έφτασε στο επίπεδο IETF draft.

Προφανώς για να υποστηριχθούν τα δίκτυα ευρείας περιοχής έπρεπε να γίνουν κάποιες αλλαγές στο πρωτόκολλο. Το WASLP εισάγει δυο νέες οντότητες, τους Advertising Agent (AA) και Broker Agent (BA). Επίσης, το δίκτυο χωρίζεται σε διάφορα domains, καθένα από τα οποία έχει τους δικούς του UAs, SAs και DAs. Σε κάθε SLP Domain (SLPD) τον ρόλο των AA τον παίζουν οι DAs και οι SAs. Οι AAs μαζεύουν πληροφορίες για τις υπηρεσίες που υπάρχουν στο domain τους και τις διαφημίζουν σε όλο το δίκτυο (δηλαδή σε όλα τα domains). Οι BAs δέχονται αυτά τα μηνύματα και συλλέγουν πληροφορίες για υπηρεσίες σε άλλα domains. Οι υπηρεσίες ομαδοποιούνται σε scopes, όπως και στο απλό SLP, μόνο που τώρα είναι πιο σημαντική η χρήση τους για να μειωθεί ο συνολικός αριθμός των μηνυμάτων.

Το σενάριο χρήσης του WASLP είναι κάπως έτσι:

- Ο UA ζητάει από τον DA μια υπηρεσία.
- Ο DA βρίσκει ότι αυτή η υπηρεσία δεν είναι διαθέσιμη τοπικά.
- Ο DA ξέρει ότι υπάρχει κάποιος τοπικός BA.
- Ο DA λειτουργεί σαν UA και ζητάει από τον BA αυτή την υπηρεσία.
- Ο BA επιστρέφει τις πληροφορίες που είχε συλλέξει για την υπηρεσία από τους AAs (είναι σημαντικό να σημειώσει που βρίσκεται η υπηρεσία).
- Ο DA επιστρέφει τις πληροφορίες στον UA.

2.11.2 Mesh-enhanced SLP (mSLP)

Το mSLP περιγράφεται στο RFC 3528 [10]. Το mSLP προβλέπει την υπάρξη πολλών ομότιμων DAs και καθορίζει την ανταλλαγή πληροφοριών μεταξύ τους. Επίσης, κάνει την διαδικασία καταχώρησης υπηρεσιών πιο εύκολη για τους SAs όταν υπάρχουν πολλοί DAs. Είναι σημαντικό ότι είναι πλήρως συμβατό με το απλό SLP και μπορεί να εγκατασταθεί σταδιακά.

Γενικά, ένας DA μπορεί να εξυπηρετεί πολλά scopes κι ένα scope μπορεί να εξυπηρετείται από πολλούς DAs. Το θέμα είναι πώς αλληλεπιδορύν οι DAs που εξυπηρετούν τα ίδια scopes, γιατί μπορεί κάποιος UA να ρωτήσει έναν DA και να μην πάρει την απάντηση που θα έδινε κι ένας άλλος DA.

Ένας DA που υποστηρίζει το mSLP ονομάζεται MDA (mesh-enhanced DA). Οι UAs τον βλέπουν σαν ένα απλό DA. Οι MDAs που εξυπηρετούν κοινά scopes φτιάχνουν μόνιμες συνδέσεις (TCP) μεταξύ τους κι ανταλλάσσουν μηνύματα για να έχουν πάντα τις ίδιες καταχωρήσεις για τα κοινά scopes. Επίσης, ένας MSA (mesh-enhanced SA) μπορεί αντί να καταχωρήσει μια υπηρεσία με όλους τους DAs, να την καταχωρήσει μόνο με ένα σύνολο MDAs των οποίων η ένωση των scopes καλύπτει τα scopes της υπηρεσίας.

2.11.3 Μηχανισμός Ειδοποιήσεων

Μερικές φορές η ύπαρξη μιας υπηρεσίας είναι απαραίτητη για έναν χρήστη, οπότε αν δεν την ανακαλύψει με την πρώτη προσπάθεια συνεχίζει να δοκιμάζει μέχρι να γίνει διαθέσιμη. Αυτός ο τρόπος λειτουργίας (που ονομάζεται και polling) σπαταλάει πολλούς πόρους του δικτύου και είναι επιθυμητό να αποφεύγεται. Μια λύση είναι να υπάρχει κάποιο σύστημα που θα ειδοποιήσει τον UA μόλις η υπηρεσία εμφανιστεί. Ένα τέτοιο σύστημα περιγράφεται στο RFC 3082 [9]. Σημειώνεται ότι ανάλογοι μηχανισμοί υπάρχουν στο πρότυπο του UPnP.

Ο μηχανισμός που περιγράφεται ανήκει στην κατηγορία των publish subscribe συστημάτων. Σε ένα μικρό δίκτυο όπου δεν υπάρχει DA δεν έχουμε στη διάθεσή μας ένα κεντρικό εξυπηρετητή για να διαχειρίζεται τα subscriptions. Έτσι αναγκαστικά οι SAs αναλαμβάνουν τον ρόλο να ειδοποιούν τους υπόλοιπους κόμβους για τα γεγονότα που σχετίζονται με αυτούς. Όταν ένας SA πρωτοεμφανίζεται σε ένα δίκτυο κάνει multicast ένα μήνυμα SrvReg που περιγράφει τις υπηρεσίες τις οποίες εξυπηρετεί. Αυτό το μήνυμα στέλνεται επαναλαμβανόμενα σε ένα διάστημα 15 δευτερολέπτων, για να εξασφαλιστεί ότι όλοι οι κόμβοι το είδαν. Υπάρχει επίσης η δυνατότητα να σταλεί νέο μήνυμα srvReg αν τα στοιχεία της υπηρεσίας αλλάξουν. Αν αργότερα ο SA αποφασίσει ότι θα σταματήσει να λειτουργεί, στέλνει multicast μήνυμα SrvDeReg. Θεωρείται ότι έτσι ειδοποιούνται όσοι ενδιαφέρονται για τα γεγονότα που αφορούν όλες τις υπηρεσίες (χωρίς όμως να υπάρχει φιλτράρισμα για το ποιες υπηρεσίες ενδιαφέρουν τον κάθε κόμβο).

Στην περίπτωση που υπάρχει κάποιος DA, τότε όταν ο UA στέλνει μήνυμα SrvRqst στον DA μπορεί να του υποδείξει ότι θέλει να ειδοποιείται για τα γεγονότα που αφορούν αυτή την υπηρεσία. Ένα subscription αποτελείται από το service type και ένα σύνολο από scopes. Ο DA ομαδοποιεί παρόμοια subscriptions και τους αναθέτει μια multicast διεύθυνση με βάση το πρωτόκολλο MADCAP (RFC 2730). Αυτή τη διεύθυνση τη στέλνει πίσω στον UA με το μήνυμα SrvRply κι εκεί πρέπει να ακούει ο UA για ειδοποιήσεις σχετικές με την υπηρεσία.

Όταν ένας SA καταχωρεί μια υπηρεσία με το μήνυμα SrvReg (ή ακόμα κι όταν ανανεώνει ή αφαιρεί μια καταχώρηση), ο DA ελέγχει αν υπάρχουν subscriptions για αυτή την υπηρεσία. Αν όντως υπάρχει σχετικό subscription, τότε ο DA στην απάντησή του με το μήνυμα SrvAck δίνει στον SA την multicast διεύθυνση που αντιστοιχεί στο subscription. Είναι υποχρέωση του SA να στείλει μήνυμα σε αυτή τη διεύθυνση για να ειδοποιήσει τους ενδιαφερόμενους.

2.12 Υλοποιήσεις του SLP

Καταρχάς αναφέρουμε ότι στο RFC 2614 [7] προτείνεται ένα API για τις διάφορες υλοποιήσεις του SLP. Μια από τις πρώτες και πιο γνωστές υλοποιήσεις είναι αυτή του OpenSLP, που διατίθεται με BSD άδεια. Πρωτοεμφανίστηκε τις αρχές του 2000 και βασιζόταν σε κώδικα που είχε προσφέρει η Caldera. Η ανάπτυξη του φαίνεται να έχει σταματήσει, αλλά το OpenSLP συνεχίζει να είναι ευρέως διαδεδομένο στην

open source κοινότητα.

Στα τέλη του 2005 εμφανίστηκε και το jSLP, που είναι μια υλοποίηση του SLP στη γλώσσα Java. Η χρήση της Java σημαίνει ότι το jSLP μπορεί να λειτουργήσει σε οποιοδήποτε σύστημα υποστηρίζει τη Java. Επίσης, υπάρχει μια έκδοση του jSLP που προορίζεται για το OSGi Framework (και είναι αυτή που χρησιμοποιήσαμε τελικά). Το jSLP αναπτύχθηκε από το Information and Communication Systems Research Group του πανεπιστημίου ETH της Ζυρίχης. Έχει κι αυτό BSD style άδεια χρήσης.

Άλλες, ανεξάρτητες υλοποιήσεις περιλαμβάνονται στα Solaris, Novell Netware, HP-UX, Axis Communication Systems, κλπ.

Βιβλιογραφία

[1] J. Veizades, E. Guttman, C. Perkins, S. Kaplan "Service Location Protocol" RFC 2165, Internet Engineering Task Force, Jun. 1997

[2] T. Howes "The String Representation of LDAP Search Filters" RFC 2254, Internet Engineering Task Force, Dec. 1997

[3] S. Thomson, T. Narten "IPv6 Stateless Address Autoconfiguration" RFC 2462, Internet Engineering Task Force, Dec. 1998

[4] E. Guttman, C. Perkins, J. Veizades, M. Day "Service Location Protocol, Version 2" RFC 2608, Internet Engineering Task Force, Jun. 1999

[5] E. Guttman, C. Perkins, J. Kempf "Service Templates and Service: Schemes" RFC 2609, Internet Engineering Task Force, Jun. 1999

[6] C. Perkins, E. Guttman "DHCP Options for Service Location Protocol" RFC 2610, Internet Engineering Task Force, Jun. 1999

[7] J. Kempf, E. Guttman "An API for Service Location" RFC 2614, Internet Engineering Task Force, Jun. 1999

[8] S. Hanna, B. Patel, M. Shah "Multicast Address Dynamic Client Allocation Protocol (MADCAP)" RFC 2730, Internet Engineering Task Force, Dec. 1999

[9] J. Kempf, J. Goldschmidt "Notification and Subscription for SLP" RFC 3082, Internet Engineering Task Force, Mar. 2001

[10] W. Zhao, H. Schulzrinne, E. Guttman, "Mesh-enhanced Service Location Protocol (mSLP)" RFC 3528, Internet Engineering Task Force, Apr. 2003

[11] S. Creshire, B. Aboba, E. Guttman "Dynamic Configuration of IPv4 Link-Local Addresses" RFC 3927, Internet Engineering Task Force, May 2005

[12] National Institute of Standards and Technology (NIST) "Digital signature standard" Technical Report NIST FIPS PUB 186, May 1994

Κεφάλαιο 3

OSGi Service Platform

3.1 Εισαγωγή

Η πλατφόρμα OSGi (Open Service Gateway Initiative) προσφέρει ένα υπολογιστικό περιβάλλον για δικτυωμένες υπηρεσίες. Με τη χρήση του OSGi σε μια δικτυωμένη συσκευή (που μπορεί να είναι είτε κάποια ενσωματωμένη (embedded) συσκευή ή ένας υπολογιστής) ο διαχειριστής έχει τη δυνατότητα να χειρίζεται τον κύκλο ζωής των επιμέρους συστατικών, ακόμα και απομακρυσμένα. Η χρήση του σε περιβάλλοντα όπου απαιτείται υψηλή αξιοπιστία κρίνεται απαραίτητη, αφού επιτρέπει την προσθήκη ή αφαίρεση συστατικών (components) χωρίς την ανάγκη επανεκκίνησης της πλατφόρμας.

Το OSGi είναι μια αρχιτεκτονική προσανατολισμένη σε υπηρεσίες (service oriented architecture). Το κάθε συστατικό προσφέρει κάποιες υπηρεσίες και χρησιμοποιεί υπηρεσίες που παρέχονται από άλλους. Αυτό οδηγεί στη μείωση του μεγέθους των εφαρμογών, αφού μοιράζονται πολύ κώδικα. Το OSGi παρέχει όλα όσα χρειάζονται για τη σωστή λειτουργία αυτού του πλαισίου, καθώς κι ένα σύνολο από προκαθορισμένες υπηρεσίες για τις πιο συνηθισμένες ενέργειες.

Το πρότυπο OSGi ορίζεται από την OSGi Alliance, στην οποία συμμετάσχουν πολλές μεγάλες εταιρείες, όπως Siemens, Sun, IBM, Motorola, Nokia, Ericsson, Vodafone, BMW κα. Το πρότυπο είναι τελείως ελεύθερο, αλλά για να συμμετάσχει κάποιος στις αποφάσεις πρέπει να είναι μέλος του OSGi Alliance. Η πιο πρόσφατη έκδοση είναι η Release 4 (Αύγουστος 2005), αν και μερικές υλοποιήσεις έχουν μείνει ακόμα στη Release 3. Το OSGi προσδιορίζεται από δύο έγγραφα:

1. Το *OSGi Service Platform Core Specification* [1], που περιγράφει τα βασικά χαρακτηριστικά, και

2. το *OSGi Service Platform Service Compendium* [2], που περιγράφει τις βασικές υπηρεσίες που πρέπει να είναι διαθέσιμες.

3.2 Αρχιτεκτονική του OSGi

Η πλατφορμά OSGi κι όλες οι εφαρμογές που τρέχουν πάνω από αυτή γράφονται στη γλώσσα Java. Αυτό έχει το επιθυμητό αποτέλεσμα ότι οι ίδιες εφαρμογές μπορούν να τρέξουν χωρίς αλλαγές σε οποιοδήποτε περιβάλλον υποστηρίζει τη γλώσσα Java. Το περιβάλλον εκτέλεσης του OSGi δεν περιορίζεται στο J2SE [3], αλλά μπορεί να τρέξει και πάνω από CDC [4], CLDC [5], MIDP [6], κα. Αυτό σημαίνει ότι μπορεί εύκολα να λειτουργήσει σε ενσωματωμένες συσκευές, όπως κινητά τηλέφωνα, set-top boxes, modems, κλπ. Ένα άλλο σημαντικό στοιχείο είναι πώς όλες οι εφαρμογές τρέχουν μέσα σε ένα μοναδικό Java Virtual Machine (JVM), γεγονός που μειώνει την απαιτούμενη μνήμη, διευκολύνοντας περαιτέρω τη χρήση σε ενσωματωμένα συστήματα.

Το OSGi είναι μια αρχιτεκτονική βασισμένη σε συνθετικά (components), τα οποία στη γλώσσα του OSGi ονομάζονται bundles. Ένα bundle διανείμεται ως ένα αρχείο jar και προσφέρει μια λειτουργικότητα στο συνολικό πλαίσιο. Ένα από τα δυνατά στοιχεία του OSGi είναι η δυναμική διαχείριση του κύκλου ζωής (life-cycle) των bundles. Ένα bundle μπορεί να εγκατασταθεί, εκκινηθεί, ανανεωθεί και σταματηθεί δυναμικά, χωρίς να σταματάει η λειτουργία του υπόλοιπου πλαισίου. Προφανώς αυτό είναι ιδιαίτερο χρήσιμο για συσκευές που χρειάζεται να λειτουργούν ασταμάτητα, γιατί μπορούν να αναβαθμιστούν χωρίς διακοπή της λειτουργίας τους.

Ίσως το πιο σημαντικό χαρακτηριστικό του OSGi είναι ο προσανατολισμός του στη χρήση υπηρεσιών (services). Κάθε bundle προσφέρει κάποιες υπηρεσίες και χρησιμοποιεί υπηρεσίες που προσφέρουν άλλα bundles. Υπάρχει ένα κεντρικό σημείο καταχώρησης των υπηρεσιών, το service registry. Το OSGi προβλέπει λειτουργίες για την αναζήτηση υπηρεσιών, καθώς και την ειδοποίηση των ενδιαφερόμενων για γεγονότα σχετικά με υπηρεσίες (όπως εμφάνιση κι εξαφάνιση). Επίσης, προωθείται ο διαχωρισμός διαπροσωπείου και υλοποίησης. Ένα αντικείμενο μπορεί να δηλωθεί ότι υλοποιεί το διαπροσωπείο μιας υπηρεσίας, αλλά υπάρχουν και πιο ευέλικτοι συνδυασμοί, όπως ένα αντικείμενο να υλοποιεί πολλές υπηρεσίες ή μια υπηρεσία να υλοποιείται από πολλά αντικείμενα.

Το πλαίσιο θα ήταν πρακτικά άχρηστο αν το κάθε bundle έπρεπε να θεωρεί ότι θα λειτουργεί σε ένα περιβάλλον χωρίς κάποιες βασικές υπηρεσίες. Ευτυχώς το πρότυπο καθορίζει πολλές υπηρεσίες που πρέπει να περιέχει μια συμβατή υλοποίηση. Οι υπηρεσίες αυτές καλύπτουν πολλούς τομείς κι επιτρέπουν την εύκολη ανάπτυξη εφαρμογών. Οι σημαντικότερες από αυτές παρουσιάζονται στη συνέχεια.

Τέλος, σημειώνουμε ότι προβλέπονται πολλοί μηχανισμοί ασφαλείας. Καταρχάς, η γλώσσα Java προσφέρει από μόνη της ένα ασφαλές περιβάλλον [7]. Επίσης, υπάρχει η δυνατότητα υπογραφής των bundles έτσι ώστε να μπορεί να αξιολογηθεί η αξιοπιστία τους. Πέρα από αυτά, μερικές από τις προβλεπόμενες υπηρεσίες μπορούν να χρησιμοποιηθούν για πιο λεπτές ρυθμίσεις σχετικές με την ασφάλεια. Η υπηρεσία Permission Admin ασχολείται με το τι επιτρέπεται να κάνει το κάθε bundle, ενώ η υπηρεσία User Admin ασχολείται με την πιστοποίηση και την εξουσιοδότηση (authentication και authorization) των χρηστών.

3.3 Βασικές υπηρεσίες του OSGi

Όπως αναφέραμε και παραπάνω, το πρότυπο του OSGi προβλέπει και την ύπαρξη κάποιων βασικών υπηρεσιών. Παρακάτω θα αναλύσουμε τις σημαντικότερες από αυτές.

3.3.1 Log Service

Όνομα κλάσης: *org.osgi.service.log.LogService*

Τα διάφορα bundles μπορούν να χρησιμοποιήσουν αυτή την υπηρεσία για την καταγραφή των γεγονότων που θεωρούν σημαντικά. Υποστηρίζεται η χρήση διαφορετικών επιπέδων levels καταγραφής, δηλαδή κάθε μήνυμα μπορεί να ανήκει σε ένα από τα επίπεδα: ERROR, WARNING, INFO ή DEBUG. Στην πιο απλή μορφή, μια καταχώρηση αποτελείται από ένα μήνυμα κι ένα συσχετιζόμενο επίπεδο. Προαιρετικά, η καταχώρηση μπορεί να περιλαμβάνει κι ένα ServiceReference που δείχνει την υπηρεσία την οποία αφορά το μήνυμα ή κι ένα Exception που σχετίζεται με το γεγονός.

Παράδειγμα χρήσης:

```
logService.log(LogService.LOG_DEBUG, "I am here");
```

Το Knopflerfish προσφέρει μια υπηρεσία, το LogRef, που απλοποιεί τη χρήση του LogService. Με το LogRef μπορεί κανείς να ορίσει πως αν χαθεί η αναφορά προς το LogService τότε τα μηνύματα θα τυπώνονται στην κονσόλα. Επίσης, το LogRef προσφέρει απλοποιημένα ονόματα συναρτήσεων, όπως

```
logRef.debug("I am here");
```

έτσι ώστε να μην χρειάζεται να δηλώνεται κάθε φορά το επίπεδο καταγραφής. Βέβαια, πρέπει να έχουμε υπόψιν μας ότι η χρήση του LogRef δεν είναι εγγυημένα συμβατή με τις υπόλοιπες υλοποιήσεις του OSGi πέραν του Knopflerfish.

3.3.2 HTTP Service

Όνομα κλάσης: *org.osgi.service.http.HttpService*

Αυτή η υπηρεσία μπορεί να λειτουργήσει ως εξυπηρετητής HTTP που σερβίρει είτε απλές σελίδες html ή Java Servlets [8]. Αυτό είναι πολύ σημαντικό, αφενός μεν γιατί το OSGi έχει σχεδιαστεί ως πλατφόρμα για δικτυακές υπηρεσίες κι αφετέρου γιατί επιτρέπει τη δημιουργία ενός γνώριμου αλληλεπιδραστικού περιβάλλοντος με τον χρήστη. Το http έχει καθιερωθεί ως η πιο διαδεδομένη δικτυακή υπηρεσία και όλοι όσοι έχουν πρόσβαση στο διαδίκτυο έχουν εξασκηθεί στη χρήση της.

Η χρήση του HTTP Service περιλαμβάνει τα συνηθισμένα βήματα. Πρώτα παίρνουμε μια αναφορά στην υπηρεσία HttpService και μετά καλούμε την μέθοδο registerServlet, όπου δίνουμε την κλάση που υλοποιεί το servlet, καθώς και το όνομα με το οποίο θέλουμε να είναι ορατό.

Ωστόσο, χρησιμοποιώντας τις δυνατότητες του OSGi μπορούμε να απλοποιήσουμε πολύ τη διαδικασία αυτή. Εφαρμόζοντας το whiteboard pattern [11], αντί να καλέσουμε την υπηρεσία HttpService καταχωρούμε το δικό μας servlet ως υπηρεσία. Φτιάχνοντας ένα wrapper γύρω από το HttpService αυτό μπορεί να ακούει για καταχωρήσεις υπηρεσιών τύπου HttpServlet και να πράττει ανάλογα. Βέβαια αυτή η συμπεριφόρα του HttpService δεν προβλέπεται από το πρότυπο, αλλά δείχνει τη δύναμη του OSGi.

3.3.3 Event Admin Service

Όνομα κλάσης: *org.osgi.service.event.EventAdmin*

Πρόκειται για μια νέα υπηρεσία που προστέθηκε για πρώτη φορά στο OSGi στο Release 4. Το OSGi περιείχε ήδη κάποιους listeners (ServiceListener, BundleListener περιγράφονται στη συνέχεια του κεφαλαίου), αλλά αναγνωρίσθηκε η αξία ύπαρξης ενός γενικότερου publish subscribe συστήματος.

Η κλάση Event αντιπροσωπεύει ένα γεγονός. Έχει ένα θέμα (topic) καθώς και ιδιότητες (properties). Οι ιδιότητες είναι ζεύγη 'ιδιότητα=τιμή'. Τα θέματα είναι οργανωμένα ιεραρχικά και σαν διαχωριστικό χρησιμοποιείται το σύμβολο '/'. Για παράδειγμα:

fully/qualified/topic/NAME

Αυτός που θέλει να δημοσιεύσει ένα Event πρέπει να χρησιμοποιήσει την υπηρεσία EventAdmin. Η υπηρεσία αυτή έχει δυο μεθόδους, τις sendEvent και postEvent. Η πρώτη παραδίδει το Event με σύγχρονο τρόπο, ενώ η δεύτερη με ασύγχρονο. Προφανώς προτιμάται ο ασύγχρονος τρόπος, γιατί δεν μπλοκάρει την κανονική ροή του προγράμματος.

Αυτός που θέλει να λαμβάνει τις ειδοποιήσεις οφείλει να καταχωρήσει μια υπηρεσία του τύπου EventHandler. Επίσης, στα χαρακτηριστικά της υπηρεσίας πρέπει να δηλώσει τα θέματα που τον ενδιαφέρουν καθώς και κάποιο πιθανό φίλτρο. Ένα παράδειγμα από το [2] φαίνεται παρακάτω:

```
public AcmeWatchDog implements Activator, EventHandler {
  final static String [] topics = new String[] {
    "org/osgi/service/log/LogEntry/LOG_WARNING",
    "org/osgi/service/log/LogEntry/LOG_ERROR" };

  public void start(BundleContext context) {
    Dictionary d = new Hashtable();
    d.put(EventConstants.EVENT_TOPICS, topics );
    d.put(EventConstants.EVENT_FILTER, "(bundle.
        symbolicName=com.acme.*)" );
    context.registerService( EventHandler.class.getName(),
        this, d );
```

```
  }

  public void stop( BundleContext context) {}

  public void handleEvent(Event event ) {
    //...
  }
}
```

3.3.4 Άλλες Υπηρεσίες

Το πρότυπο του OSGi ορίζει κι άλλες υπηρεσίες, αλλά προφανώς δεν γίνεται να τις παρουσιάσουμε όλες, γιατί θα ξεφεύγαμε από τους σκοπούς αυτού του εγγράφου. Ενδεικτικά αναφέρουμε ότι προβλέπεται η ύπαρξη υπηρεσιών για UPnP, XML parser καθώς και απομακρυσμένη πρόσβαση. Επίσης, αφού το OSGi προορίζεται για χρήση και σε ενσωματωμένες συσκευές, υπάρχουν υπηρεσίες για πρόσβαση στις συσκευές και έχει εισαχθεί και το Connector framework [9] από τη J2ME [10]. Ακόμα παρέχονται υπηρεσίες για αυτόματη ρύθμιση κάποιων παραμέτρων, για το συντονισμό της ασφάλειας, για την αποθήκευση των προτιμήσεων των χρηστών, κα.

3.4 Προγραμματισμός του OSGi

Στη συνέχεια θα παρουσιάσουμε πώς προγραμματίζεται μια εφαρμογή για το OSGi. Όπως ήδη αναφέραμε, η εφαρμογή πακετάρεται σε ένα αρχείο jar. Μέσα στο jar υπάρχει το αρχείο Manifest.mf που περιέχει σημαντικές πληροφορίες που πρέπει να γνωρίζει το framework για την εφαρμογή. Στο Knopflerfish το Manifest.mf παράγεται αυτόματα από το ant και χρειάζεται να ορίσουμε μόνο μερικές παραμέτρους, σημαντικότερη από τις οποίες είναι ποιες άλλες υπηρεσίες από τρίτα bundles θα χρησιμοποιηθούν. Άλλες πληροφορίες που περιέχονται στο Manifest.mf είναι οι υπηρεσίες που εξάγει το bundle, που βρίσκεται ο BundleActivator, η έκδοση και ο δημιουργός του bundle κα.

3.4.1 Συνήθεις πρακτικές

Σημειώσαμε ήδη πως το OSGi προωθεί τον διαχωρισμό διαπροσωπείου και υλοποίησης. Είναι πολύ συνηθισμένο να τοποθετούνται τα interfaces σε χωριστό package και η υλοποίηση να τοποθετείται σε άλλο package που να τελειώνει σε .impl (από το implementation). Κάθε bundle πρέπει να περιέχει μια κλάση που να υλοποιεί το interface BundleActivator. Αυτή η κλάση τοποθετείται συχνά σε αρχείο με όνομα Activator.java. Κανονικά πρέπει να δηλώνεται στο Manifest.mf ποια κλάση είναι ο Activator, αλλά στο Knopflerfish το ant βρίσκει αυτόματα την κλάση.

3.4.2 BundleActivator

Αυτή είναι η κλάση που θα κληθεί όταν εκκινηθεί ή σταματήσει το bundle (οι μέθοδοι start και stop αντίστοιχα). Κατά την εκκίνηση πιθανώς πρέπει να γίνουν πολλές ενέργειες, αλλά είναι επιθυμητό να μην καθυστερεί πολύ η κλήση της start. Για το λόγο αυτό, συχνά η start κάνει μόνο τα απαραίτητα και δημιουργεί threads για τις υπόλοιπες εργασίες.

Μια από τις πιο συχνές λειτουργίες του Activator είναι να αποθηκεύσει το αντικείμενο BundleContext που του παρέχεται όταν καλείται η start. Το αντικείμενο αυτό είναι άκρως απαραίτητο, αφού ουσιαστικά αποτελεί την δίοδο προς τις λειτουργίες του framework. Πολλές φορές δηλώνεται ως static, γιατί έτσι δεν χρειάζεται να περνάει σαν παράμετρος σε όλες τις υπόλοιπες κλάσεις του bundle (αφού θα έχουν πρόσβαση σε αυτό ως πχ Activator.bc). Ακόμα, στη μέθοδο start μπορεί να εισάγονται και να εξάγονται υπηρεσίες, αλλά θα δούμε παρακάτω πως γίνεται αυτό.

Όταν ένα bundle σταματάει δεν χρειάζεται να αποδεσμεύσει τις υπηρεσίες που έχει εισάγει, γιατί αυτό γίνεται αυτόματα από το framework. Όμως είναι απαραίτητο να καθαρίσει τα τυχόν threads που έχει δημιουργήσει και να μηδενίσει τις διάφορες static μεταβλητές που υπάρχουν.

Στο 3.1 φαίνεται ο σκελετός ενός απλού BundleActivator.

```
package org.acme.test.impl;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

  static BundleContext bc = null;

  public void start(BundleContext context) throws Exception
      {
    bc = context;
    ...
  }

  public void stop(BundleContext context) throws Exception
     {
    bc = null;
    ...
  }
}
```

Listing 3.1: Activator example

3.4.3 Εισαγωγή μιας υπηρεσίας

Υπάρχουν 3 τρόποι να χρησιμοποιηθούν υπηρεσίες από άλλα bundles, καθένας με τα πλεονεκτήματα και τα μειονεκτήματά του. Σε κάθε περίπτωση, αναζητούμε μια υπηρεσία με βάση το όνομα του interface το οποίο υλοποιεί.

Οφείλουμε να θυμόμαστε ότι το OSGi Service Platform είναι ένα δυναμικό περιβάλλον, όπου υπηρεσίες μπορούν να εμφανιστούν και να εξαφανιστούν. Αυτό έχει ιδιαίτερη σημασία για τον πρώτο τρόπο, τον οποίο περιγράφουμε παρακάτω.

Ο απλός τρόπος

Παράδειγμα αυτού του τρόπου φαίνεται στο Listing 3.2.

Εδώ φαίνεται και η σημασία του αντικειμένου BundleContext (που έχει το όνομα bc στον κώδικα που φαίνεται). Μέσω αυτού του αντικειμένου αναζητάμε μια υπηρεσία με βάση το όνομά της και παίρνουμε πίσω ένα ServiceReference και μετά πάλι μέσω αυτού παίρνουμε τελικά το αντικείμενο που υλοποιεί την υπηρεσία που θέλουμε.

Όταν καλούμε τη μέθοδο getServiceReference δεν είναι σίγουρο ότι θα πάρουμε κάποια απάντηση, γιατί μπορεί να μην είναι διαθέσιμη ζητούμενη υπηρεσία (γι' αυτό ελέγχουμε αν το αποτέλεσμα είναι null). Ακόμα, είναι δυνατό να υπάρχουν περισσότερες υλοποιήσεις για την υπηρεσία αυτή, στην οποία περίπτωση το framework διαλέγει αυτόματα ποια θα επιστρέψει. Αν ενδιαφερόμαστε να βρούμε όλες τις υλοποιήσεις της υπηρεσίας μπορούμε να χρησιμοποιήσουμε τη μέθοδο getServiceReferences (προσοχή στο τελικό 's'). Επίσης, στην τελευταία γραμμή φαίνεται πώς μπορούμε να αποδεσμεύσουμε μια υπηρεσία, αν και όπως είπαμε αυτό γίνεται αυτόματα όταν σταματάει το δικό μας bundle.

```
ServiceReference ref = bc.getServiceReference(MyService.
    class.getName());
if (ref != null) {
  MyService service = (MyService) bc.getService(ref);
}
bc.ungetService(ref);
```

Listing 3.2: Simple ServiceReference

Ας σημειωθεί ότι αντί να γράφουμε το πλήρες όνομα μιας κλάσης είναι πιο συνηθισμένο να γράφουμε '<MyClass>.class.getName()'.

Παρόλο που αυτός ο τρόπος είναι ο πιο απλός, τα μειονεκτήματά του είναι εμφανή αμέσως. Καταρχάς, αν δεν λάβουμε απάντηση κατευθείαν δεν υπάρχει κάποιος τρόπος να εισάγουμε την υπηρεσία αργότερα, όταν θα είναι διαθέσιμη. Οι συνεχείς δοκιμές (polling) δεν είναι αποδοτικός τρόπος κι αλλοιώνει την κανονική ροή του προγράμματος. Αυτό από μόνο του είναι σημαντικός λόγος για τη χρήση των άλλων δύο τρόπων. Ένα άλλο, πιο λεπτό πρόβλημα είναι ότι από τη στιγμή που θα λάβουμε το ServiceReference μέχρι να δοκιμάσουμε να πάρουμε το αντικείμενο που υλοποιεί την υπηρεσία μπορεί η υπηρεσία να πάψει να είναι διαθέσιμη. Αν και εύκολα

μπορούμε να υποθέσουμε ότι αυτή η περίπτωση είναι σπάνια, δεν μπορούμε να την αγνοήσουμε.

Χρήση του ServiceListener

Μια λύση στα παραπάνω προβλήματα είναι να χρησιμοποιήσουμε το ServiceListener, που είναι ένα event listener που καλείται όποτε υπάρχει κάποια αλλαγή στις υπηρεσίες.

Ένα απλό παράδειγμα ενός ServiceListener φαίνεται στο Listing 3.3.

```
class MyListener implements ServiceListener {
  public void serviceChanged(ServiceEvent event) {
    switch (event.getType()) {
    case ServiceEvent.REGISTERED: ...
    case ServiceEvent.MODIFIED: ...
    case ServiceEvent.UNREGISTERING: ...
    }
  }
}
```

Listing 3.3: Simple ServiceListener

Εφόσον δημιουργήσουμε τον ServiceListener πρέπει να τον καταχωρήσουμε στο framework, όπως φαίνεται παρακάτω:

```
MyListener listener = new MyListener();
String filter = "(objectclass=" + MyService.class.getName()
    + ")";
bc.addServiceListener(listener, filter);
```

Listing 3.4: Add a ServiceListener

Δηλαδή φτιάχνουμε ένα αντικείμενο τύπου ServiceListener κι ένα φίλτρο που ορίζει για ποιες υπηρεσίες θα καλείται ο listener. Ουσιαστικά πρόκειται για ένα publish subscribe σύστημα, όπως περιγράφτηκε στα προηγούμενα κεφάλαια.

Η χρήση του ServiceListener αντιμετωπίζει επιτυχώς το θέμα της δυναμικής φύσης των υπηρεσιών. Το πρόβλημα είναι πως αν ServiceListener καταχωρηθεί *μετά* την εμφάνιση της υπηρεσίας τότε το πρόγραμμά μας δεν θα ειδοποιηθεί για την ύπαρξη της υπηρεσίας, παρά μόνο αν συμβεί κάποιο γεγονός σχετικό με την υπηρεσία. Αυτό οδήγησε στη δημιουργία του ServiceTracker.

Χρήση του ServiceTracker

Ο ServiceTracker σχεδιάστηκε για να αντιμετωπίσει τα προβλήματα του ServiceListener. Δηλαδή ο ServiceTracker αυτόματα εντοπίζει τις υπηρεσίες, χωρίς να χρειάζεται να περιμένει να συμβεί κάποιο γεγονός στις υπηρεσίες. Επίσης, παρέχει

μερικές παραπάνω λειτουργίες, όπως να παρέχει το αντικείμενο της υπηρεσίας αν αυτή υπάρχει ή να μπλοκάρει το πρόγραμμα μέχρι να εμφανιστεί υπηρεσία.

Ένας ServiceTracker μπορεί να δημιουργηθεί με έναν από τους παρακάτω τρόπους:

```
ServiceTracker(BundleContext, String,
    ServiceTrackerCustomizer)
ServiceTracker(BundleContext, Filter,
    ServiceTrackerCustomizer)
ServiceTracker(BundleContext, ServiceReference,
    ServiceTrackerCustomizer)
```

Σε κάθε περίπτωση η πρώτη παράμετρος είναι το κλασικό αντικείμενο BundleContext και η δεύτερη όριζει ποιες υπηρεσίες θα παρακολουθεί ο ServiceTracker. Η τρίτη παράμετρος είναι ένα αντικείμενο τύπου ServiceTrackerCustomizer που καθορίζει πώς θα αντιδρά ο ServiceTracker στις αλλαγές που συμβαίνουν στις υπηρεσίες. Παρακάτω φαίνεται ένα παράδειγμα ενός ServiceTrackerCustomizer:

```
class MyServiceTrackerCustomizer implements
    ServiceTrackerCustomizer {

  private BundleContext bc;

  public MyServiceTrackerCustomizer(BundleContext context)
      {
    bc = context;
  }

  public Object addingService(ServiceReference ref) {
    return bc.getService(ref);
  }

  public void modifiedService(ServiceReference ref, Object
      service) {
    ...
  }

  public void removedService(ServiceReference ref, Object
      service) {
    ...
  }
}
```

Όπως φαίνεται υπάρχει μια μέθοδος για κάθε γεγονός που μπορεί να αφορά μια υπηρεσία. Η χρήση του ServiceTracker φαίνεται παρακάτω

```
MyServiceTrackerCustomizer mstc = new
    MyServiceTrackerCustomizer(bc);
ServiceTracker tracker = new ServiceTracker(bc, MyService.
    class.getName(), mstc);
tracker.open();

tracker.waitForService(2000);
MyService service = (MyService) tracker.getService();
```

Η λειτουργία του ServiceTracker ξεκινάει με την κλήση tracker.open(). Με την μέθοδο waitForService(long) το πρόγραμμα μπλοκάρει μέχρι να εμφανιστεί η υπηρεσία ή να λήξει το χρονικό διάστημα που ορίζεται από τη μια παράμετρο (σε milliseconds). Το πρότυπο συνιστά ότι αυτή η μέθοδος δεν πρέπει να καλείται από τον BundleActivator γιατί αυτός οφείλει να τελειώνει γρήγορα. Τέλος, με την μέθοδο getService() παίρνουμε το αντικείμενο που υλοποιεί την υπηρεσία.

3.4.4 Εξαγωγή μιας υπηρεσίας

Η διαδικασία δήλωσης μιας υπηρεσίας για να μπορεί να χρησιμοποιηθεί από τα υπόλοιπα bundles είναι πολύ απλή και φαίνεται στη συνέχεια. Είναι συχνό να δηλώνονται οι υπηρεσίες από τον Activator.

```
ServiceRegistration reg;
MyServiceImpl service = new MyServiceImpl();
Hashtable props = new Hashtable();
reg = bc.registerService(MyService.class.getName(), service
    , props);
reg.unregister();
```

Όπως φαίνεται, χρησιμοποιείται και πάλι το αντικείμενο bundlecontext και πιο συγκεκριμένα η μέθοδος registerService. Η πρώτη παράμετρος της μεθόδου δηλώνει ποια υπηρεσία υλοποιεί το αντικείμενο που εξάγουμε, η δεύτερη είναι το ίδιο το αντικείμενο και η τρίτη είναι τα χαρακτηριστικά (properties) της υλοποίησής μας. Τα χαρακτηριστικά μπορεί να είναι κενά, όπως στο παράδειγμα.

Σημειώνουμε ότι το αντικείμενο που δηλώνουμε (της κλάσης MyServiceImpl) ανήκει σε διαφορετική κλάση από την υπηρεσία που υλοποιεί (κλάση MyService). Αυτό είναι σύμφωνο με την νοοτροπία του OSGi που προωθεί το διαχωρισμό API και υλοποίησης. Επίσης, ένα αντικείμενο μπορεί να υλοποιεί παραπάνω από μια υπηρεσία, στην οποία περίπτωση η πρώτη παράμετρος της μεθόδου registerService είναι ένας πίνακας με τα ονόματα όλων των υπηρεσιών που υλοποιεί.

Η μέθοδος registerService επιστρέφει ένα αντικείμενο της κλάσης ServiceRegistration. Αν αργότερα θελήσουμε να αφαιρέσουμε την υπηρεσία πρέπει να αποθηκεύσουμε αυτό το αντικείμενο, γιατί μέσω αυτού γίνεται η αφαίρεση της υπηρεσίας (όπως φαίνεται στο παράδειγμα). Με το ίδιο αντικείμενο μπορούμε να αλλάξουμε

τα χαρακτηριστικά μιας υπηρεσίας ακόμα και μετά τη δήλωσή της, με τη μέθοδο setProperties(Properties props).

Άλλες δυνατότητες

Είναι ενδιαφέρον ότι ένα bundle μπορεί διαχειριστεί τα υπόλοιπα bundles (εφόσον έχει βέβαια τα απαραίτητα δικαιώματα). Παρακάτω φαίνεται πώς μπορεί ένα bundle να εγκαταστήσει και να εκκινήσει ένα άλλο bundle.

```
String location = "jars/mybundle/mybundle.jar";
Bundle bundle = bc.installBundle(location);
bundle.start();
```

Η κλάση Bundle αντιπροσωπεύει προφανώς ένα bundle. Κάθε bundle έχει κι ένα μοναδικό αναγνωριστικό νούμερο. Μπορούμε να χειριστούμε τα bundles και μέσω αυτού του νούμερου.

```
Bundle bundle = bc.getBundle(23);
bundle.stop();
```

Μια άλλη ενδιαφέρουσα δυνατότητα στο OSGi είναι η χρήση της κλάσης BundleListener για να παρακολοθούμε τον κύκλο ζωής των διάφορων bundles. Λειτουργεί όπως η κλάση ServiceListener, δηλαδή η μέθοδος bundleChanged καλείται όποτε αλλάζει η κατάσταση του bundle. Ένας σκελετός της δυνατότητας αυτής φαίνεται παρακάτω.

```
class MyBundleListener implements BundleListener {
  public void bundleChanged(BundleEvent event) {
    switch (event.getType()) {
    case BundleEvent.STARTED: ...
    case BundleEvent.STOPPED: ...
    ...
    }
  }
}

MyBundleListener mbl = new MyBundleListener;
bc.registerBundleListener(mbl);
```

3.5 Εφαρμογές του OSGi

Στη συνέχεια παρουσιάζονται μερικοί από τους τομείς όπου έχει χρησιμοποιηθεί με επιτυχία το OSGi Service Platform.

Το Eclipse [12] είναι ένα περιβάλλον ανάπτυξης εφαρμογών. Αρχικά αναπτύχθηκε από την IBM, αλλά αργότερα διατέθηκε με άδεια ελεύθερου λογισμικού. Το

Eclipse είναι γραμμένο σε Java και στις πρώτες εκδόσεις υπήρχε υποστήριξη μόνο για ανάπτυξη εφαρμογών Java. Τώρα πια το Eclipse είναι μια επεκτάσιμη πλατφόρμα και υπάρχουν plugins για τις περισσότερες γλώσσες προγραμματισμού. Ένας από τους στόχους των προγραμματιστών του Eclipse ήταν να επιτρέπεται η δυναμική εισαγωγή plugins χωρίς να απαιτείται επανεκκίνηση της πλατφόρμας. Για να αντιμετωπίσουν το θέμα αυτό, κατά τη μετάβαση από την έκδοση 2 στην έκδοση 3 το υπόβαθρο του Eclipse αναδιοργανώθηκε για να βασίζεται στο OSGi.

Το OSGi μπορεί να χρησιμοποιηθεί επίσης στα ηλεκτρονικά συστήματα των αυτοκινήτων. Η BMW (που είναι και μέλος του OSGi Alliance) το χρησιμοποιεί ήδη στις σειρές 5 και 6, τόσο για τον έλεγχο των διαφόρων υποσυστημάτων όσο και για το σύστημα ψυχαγωγίας. Η δυνατότητα λειτουργίας σε ενσωματωμένα συστήματα σε συνδυασμό με την ευκολία αναβάθμισης και προσθήκης νέων υπηρεσιών καθιστά το OSGi ιδανικό για την περίπτωση αυτή.

Ακόμα, πολλές προσπάθειες για το έξυπνο σπίτι (smart home) βασίζονται στο OSGi. Για παράδειγμα, η πρωτοβουλία Home Genie της Shell χρησιμοποιεί ένα gateway που ελέγχει κάμερες, λάμπες, μετρητές, κλπ. Το gateway τρέχει το OSGi, προσφέροντας τη δυνατότητα απομακρυσμένης διαχείρισης.

3.6 Υλοποιήσεις του OSGi

Η OSGi Alliance ορίζει το πρότυπο του OSGi Service Platform αλλά δεν παρέχει κάποια υλοποίηση. Οποιοσδήποτε είναι ελεύθερος να διαβάσει το πρότυπο και να δημιουργήσει μια συμβατή υλοποίηση. Η κοινότητα του ελεύθερου λογισμικού γρήγορα ενστερνίστηκε το καινούργιο πρότυπο, ενώ υπάρχουν και πολλές κλειστές υλοποιήσεις.

Το Oscar [13] ήταν μια από τις πρώτες (Νοέμβριος 2002) υλοποιήσεις σε ελεύθερο λογισμικό. Είναι σχεδόν πλήρως συμβατό με το Release 3, αλλά η ανάπτυξη του φαίνεται να έχει σταματήσει. Ένα από τα σημαντικά στοιχεία που εισήγαγε το Oscar ήταν το Oscar Bundle Repository (OBR) [14], δηλαδή μια online βιβλιοθήκη από bundles τα οποία ο χρήστης μπορεί να τα εγκαταστήσει αυτόματα.

Το Knopflerfish [15] είναι άλλη μια ελεύθερη υλοποίηση (BSD-style άδεια χρήσης). Το Knopflerfish αναπτύσσεται με την υποστήριξη της Gatespace Telematics, η οποία παρέχει και δικιά της, εμπορική υλοποίηση. Η έκδοση 2 του Knopflerfish που κυκλοφόρησε πρόσφατα (28 Ιουνίου 2006) είναι πλήρως συμβατή με το Release 4. Το Knopflerfish υποστηρίζει το OBR, ενώ έχουν αναπτύξει και πολλά δικά τους bundles. Ένα τέτοιο bundle είναι το Knopflerfish OSGi Desktop, το οποίο είναι ένα εύχρηστο γραφικό περιβάλλον που δίνει στο χρήστη τη δυνατότητα να εγκαταστήσει, να εκκινήσει ή να σταματήσει bundles με εύκολο τρόπο.

Όπως έχει ήδη αναφερθεί, το Eclipse βασίζεται στο OSGi. Έχουν την δικιά τους υλοποίηση, την οποία έκαναν διαθέσιμη υπό το όνομα Equinox [16]. Το Equinox είναι κι αυτό πλήρως συμβατό με το Release 4. Το θέμα είναι πώς το Equinox είναι πολύ στενά συνδεδεμένο με το Eclipse, γιατί δεν αναπτύχθηκε εξ αρχής ως καθαρή OSGi υλοποίηση. Όπως αναφέρουν ρητά οι δημιουργοί στη σελίδα του, θα αναπτύ-

ξουν οποιαδήποτε μη-προτυποποιημένο χαρακτηριστικό θεωρούν απαραίτητο για τη λειτουργία των εφαρμογών OSGi.

Πρόσφατα και το Apache Foundation [17] ξεκίνησε ένα νέο project, με την ονομασία Felix [18], για τη δημιουργία δικής τους υλοποίησης που θα είναι συμβατή με το Release 4. Το έργο δεν έχει ολοκληρωθεί ακόμα, αλλά μάλλον θα πρέπει να περιμένουμε κάτι το αξιόλογο.

Υπάρχουν επίσης πολλές εμπορικές υλοποιήσεις του OSGi. Ενδεικτικά αναφέρουμε ότι οι παρακάτω εταιρίες προσφέρουν δικές τους υλοποιήσεις: IBM, Siemens, Prosyst, Gatespace, Atinav, Espial. Μια εμπορική εταιρία που προσφέρει υποστήριξη για OSGi εφαρμογές είναι η aQute [19]. Την αναφέρουμε χωριστά γιατί ο δημιουργός της, Peter Kriens, συνεργάζεται με το OSGi Alliance και αρθρογραφεί συχνά για διάφορα σχετικά θέματα.

Βιβλιογραφία

[1] "OSGi Service Platform Core Specification"
http://www.osgi.org/resources/spec_download.asp, Aug. 2005
OSGi Alliance

[2] "OSGi Service Platform Service Compendium"
http://www.osgi.org/resources/spec_download.asp, Aug. 2005
OSGi Alliance

[3] Java 2 Standard Edition
http://java.sun.com/javase/index.jsp
Sun Developer Network

[4] Connected Device Configuration (CDC)
http://java.sun.com/products/cdc/
Sun Developer Network

[5] Connected Limited Device Configuration (CLDC)
http://java.sun.com/products/cdlc/
Sun Developer Network

[6] Mobile Information Device Profile (MIDP)
http://java.sun.com/products/midp/
Sun Developer Network

[7] "Java Security"
J. Steven Fritzinger, Marianne Mueller Sun Microsystems

[8] Java Servlet Technology
http://java.sun.com/products/servlet/index.html
Sun Developer Network

[9] javax.microedition.io whitepaper
http://wireless.java.sun.com/midp/chapters/j2mewhite/chap13.pdf
Sun Developer Network

[10] Java 2 Micro Edition
http://java.sun.com/javame/index.jsp
Sun Developer Network

[11] “Listener Pattern Considered Harmful: The "Whiteboard" Pattern”
P. Kriens, BJ Hargrave
http://www.osgi.org/documents/osgi_technology/whiteboard.pdf

[12] Eclipse Foundation
http://www.eclipse.org

[13] Oscar - An OSGi Framework Implementation
http://oscar.objectweb.org/
Richard S. Hall

[14] Oscar Bundle Repository
http://oscar-osgi.sourceforge.net/

[15] Knopflerfish OSGi
http://www.knopflerfish.org/

[16] Equinox
http://www.eclipse.org/equinox/

[17] Apache Software Foundation
http://www.apache.org/

[18] Apache Felix Project
http://incubator.apache.org/felix/

[19] aQute and OSGi/Java Technologies
http://www.aqute.biz/

Κεφάλαιο 4

Απομακρυσμένες κλήσεις και SOAP

4.1 Ιστορία

Η ιδέα της χρήσης υπηρεσιών που παρέχει ένας απομακρυσμένος κόμβος είναι σχεδόν τόσο παλιά όσο και η δικτύωση των υπολογιστών. Ήδη από το 1976 (όταν υπήρχε μόνο το ARPANET) στο RFC 707 [1] είχε σκιαγραφηθεί η χρήση πόρων από άλλους δικτυωμένους δικτυωμένους κόμβους. Στη συνέχεια παρουσιάζουμε μερικά από τα σημαντικότερα πρωτόκολλα που έχουν προταθεί από τότε, μέχρι να φτάσουμε στο SOAP.

4.1.1 Βασικές έννοιες εν συντομία

Ίσως το πιο σημαντικό που πρέπει να γνωρίζουμε για μια απομακρυσμένη υπηρεσία είναι το διαπροσωπείο (interface) της. Δε γινεται να καλέσουμε μια ρουτίνα χωρίς να γνωρίζουμε τον αριθμό και το είδος των παραμέτρων που ζητάει. Παλαιότερα θα μπορούσαμε να υποθέσουμε ότι αυτά τα στοιχεία είναι γνωστά σε αυτόν που κάνει την κλήση, αλλά τώρα πια δεν είναι αποδεκτό αυτό. Για το λόγο αυτό αναπτύχθηκαν γλώσσες που περιγράφουν το διαπροσωπείο των υπηρεσιών και ονόμαζονται Interface Description Languages (IDL).

Επίσης, είναι ιδιαίτερα σημαντικός ο τρόπος με τον οποίο μεταφέρονται τα δεδομένα πάνω από το δίκτυο. Διαφορετικές γλώσσες αναπαριστούν διαφορετικά τους τύπους δεδομένων, αλλά και διαφορετικές αρχιτεκτονικές υπολογιστών έχουν ασυμβατότητες (πχ little-endian και big-endian μηχανήματα). Όλα τα πρωτόκολλα απομακρυσμένων κλήσεων περιγράφουν την αναπαράσταση των δεδομένων που στέλνονται πάνω από το δίκτυο, έτσι ώστε να μην υπάρχουν ασυμβατότητες. Η διαδικασία μετατροπής των δεδομένων στην αναπαράσταση αυτή λέγεται συνήθως marshalling ή serialization.

Ένα άλλο συχνό φαινόμενο είναι η χρήση των stubs. Τα stubs είναι μικρά

προγράμματα που έχουν το ίδιο διαπροσωπείο με την απομακρυσμένη υπηρεσία και κάθε κλήση σε αυτά μεταφράζεται σε κλήση στην υπηρεσία αυτή. Με αυτό το τρόπο ο πελάτης μπορεί να καλέσει ένα stub όπως θα καλούσε ένα τοπικό πρόγραμμα, χωρίς να ασχολείται με τις λεπτομέριες του πρωτοκόλλου. Συνήθως τα stubs μπορούν να δημιουργηθούν αυτόματα από την περιγραφή της υπηρεσίας σε IDL.

Ένα τελευταίο στοιχείο είναι η ανακάλυψη των υπηρεσιών. Όταν αναφερόμαστε σε απομακρυσμένες κλήσεις πρέπει να γνωρίζουμε που βρίσκεται το άλλο άκρο. Μερικά πρωτόκολλα προβλέπουν την ανακάλυψη του άλλου άκρου σύμφωνα με το είδος της υπηρεσίας, ενώ κάποια άλλα βασίζονται στη χρήση τρίτων πρωτοκόλλων (όπως πχ είναι το SLP).

4.1.2 RPC (Remote Procedure Call)

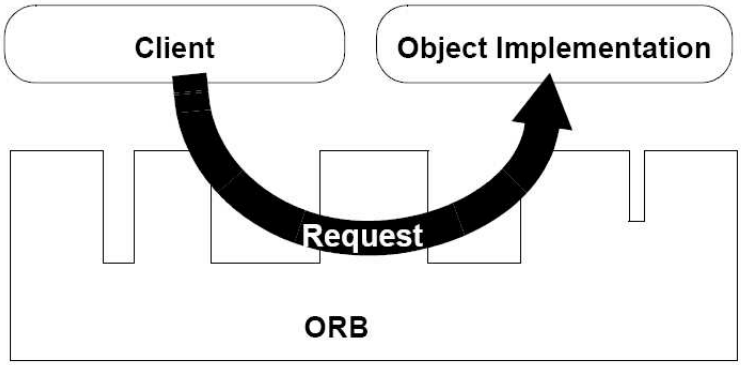
Το RPC ([3], [6]) ήταν από τα πρώτα πρωτόκολλα που επέτρεπε την κλήση ρουτίνων που βρίσκονται σε απομακρυσμένους κόμβων. Έγινε δημοφιλές με την υλοποίηση της Sun για UNIX μηχανήματα (αυτή η υλοποίηση αναφέρεται συνήθως ως Sun RPC και μερικές φορές ως ONC RPC). Το RPC αποτελεί βασικό συστατικό του NFS (Network File System), που επίσης αναπτύχθηκε από τη Sun.

Το RPC βασίζεται στο μοντέλο εξυπηρετητή-πελάτη. Ο πελάτης στέλνει στον εξυπηρετητή ένα μήνυμα που περιέχει την επιθυμητή ρουτίνα και τις παραμέτρους και ο εξυπηρετητής απαντάει με ένα μήνυμα που περιέχει το αποτέλεσμα. Τα μηνύματα στο RPC είναι συνήθως σύγχρονα, αν και δεν υπάρχει κάποιος περιορισμός κατά των ασύγχρονων κλήσεων. Επίσης, μπορεί να χρησιμοποιηθεί είτε το TCP ή το UDP.

Τα δεδομένα στα μηνύματα κωδικοποιούνται σύμφωνα με το XDR ([2], [7]). Το XDR αναλαμβάνει την αναπαράσταση των δομών δεδομένων σε μορφή ανεξάρτητη του είδους του υπολογιστή και λύνει θέματα όπως τη σειρά των bytes.

Στο RPC τα προγράμματα, η έκδοσή τους και οι ρουτίνες αναγνωρίζονται με έναν αντίστοιχο αριθμό, ο οποίος πρέπει να είναι μοναδικός. Το RPC δεν προβλέπει κάποιο μηχανισμό μετατροπής αυτών των αριθμών σε διεύθυνση δικτύου, οπότε η διεύθυνση (ή το όνομα) του εξυπηρετητή πρέπει να είναι γνωστή στο χρήστη. Η μόνη πρόβλεψη που υπάρχει είναι για τη θύρα, που δε χρειάζεται να βρίσκεται σε σταθερό αριθμό. Για αυτό το σκοπό χρησιμοποιείται μια άλλη RPC υπηρεσία, ο portmapper [8] που τρέχει σε σταθερή θύρα (111). Ο portmapper τρέχει σε κάθε μηχάνημα και ένας εξυπηρετητής RPC πρέπει να δηλώνει στον τοπικό portmapper ποιες θύρες χρησιμοποιεί. Ο πελάτης ξέρει τη διεύθυνση του εξυπηρετητή και τη θύρα του portmapper, συνδέεται στον portmapper, μαθαίνει ποια θύρα χρησιμοποιεί ο εξυπηρετητής και μετά κάνει την RPC κλήση.

Τώρα πια το RPC χρησιμοποιείται πρακτικά μόνο στο NFS. Ιστορικά παραμένει ένα σημαντικό πρωτόκολλο, αλλά υπολείπεται των νεότερων υλοποιήσεων που έχουν αντικειμενοστραφή χαρακτηριστικά.



Σχήμα 4.1: Λειτουργία του ORB

4.1.3 CORBA

Το Common Object Request Broker Architecture (CORBA) [10] είναι ένα ανοικτό πρότυπο που προσδιορίζεται από Object Management Group (OMG) [11]. Το OMG δημιουργήθηκε το 1989 με 12 μέλη και τώρα έχει περίπου 600 μέλη. Εκτός από το CORBA προσδιορίζει κι άλλα πρότυπα, με πιο σημαντικό το UML [12]. Το OMG δεν παράγει κάποια υλοποίηση· ο ρόλος του είναι μόνο να προσδιορίσει τα πρότυπα.

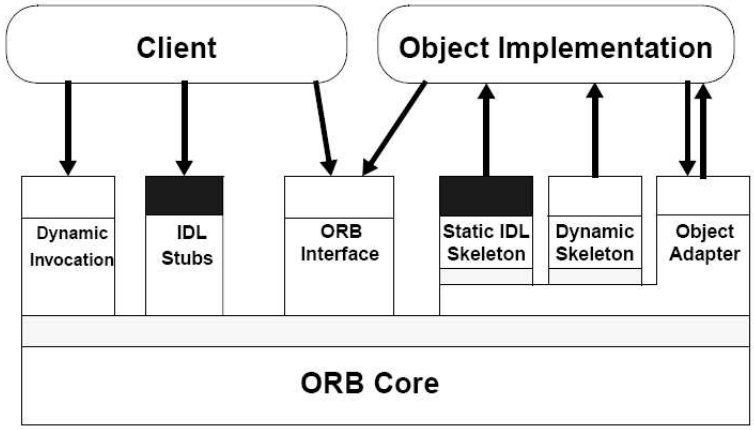
Καταρχάς σημειώνουμε ότι το CORBA είναι αντικειμενοστραφές, γεγονός που επεκτείνει σημαντικά το απλό μοντέλο του RPC. Στην καρδιά του CORBA βρίσκονται οι Object Request Brokers (ORBs) που αναλαμβάνουν την παρουσίαση των απομακρυσμένων αντικειμένων ως τοπικά (Σχήμα 4.1). Σε κάθε άκρο υπάρχει ένας ORB και οι ORBs επικοινωνούν μεταξύ τους χρησιμοποιώντας το πρωτόκολλο IIOP.

Το OMG ορίζει τη δικιά του IDL, την OMG IDL. Κάθε αντικείμενο έχει τη δικιά του περιγραφή σε IDL, αλλά η υλοποίηση μπορεί να γίνει σε οποιαδήποτε γλώσσα[1]. Στην πλευρά του πελάτη ένα stub υλοποιεί το διαπροσωπείο, ενώ στην πλευρά του εξυπηρετητή παράγεται ένα skeleton το οποίο και πρέπει να υλοποιεί το εν λόγω αντικείμενο. Η παραπάνω περιγραφή απεικονίζεται στο Σχήμα 4.2. Στο σχήμα περιλαμβάνονται κι άλλα στοιχεία, αλλά τα παραλείπουμε χάριν συντομίας.

Ένα αντικείμενο μπορεί να έχει πολλά instances. Για παράδειγμα, αν το αντικείμενο αντιπροσωπεύει ένα καλάθι με ψώνια, το καλάθι του κάθε πελάτη θα αντιστοιχεί σε ένα διαφορετικό instance. Όταν καλούμε μια μέθοδο πρέπει να προσδιορίσουμε ποιο instance ζητάμε. Το ORB προσδιορίζει αν το εν λόγω instance είναι τοπικό ή απομακρυσμένο κι αν είναι απομακρυσμένο αναλαμβάνει να το καλέσει με τρόπο διαφανή ως προς τον χρήστη.

Προς το τέλος της δεκαετίας του '90 το CORBA ήταν πολύ δημοφιλές. Τώρα πια η δημοτικότητα έχει μειωθεί κατακόρυφα. Ένας από τους λόγους αυτής της πτώσης ήταν η πολύ μεγάλη πολυπλοκότητα που περιλαμβάνει ο προγραμματισμός

[1] Αρχικά υποστηριζόταν μόνο η C++, αλλά μετά προστέθηκαν η Java κι άλλες γλώσσες.



Σχήμα 4.2: Αρχιτεκτονική του CORBA

σε CORBA. Επίσης, εμφανίστηκαν νέες τεχνολογίες που κατάφεραν να κερδίσουν το ενδιαφέρον του κόσμου. Μια ανάλυση των λόγων της παρακμής του CORBA μπορεί να βρεθεί στο [13].

4.1.4 DCOM

Το DCOM (Distributed Component Object Model) [14] ήταν η απάντηση της Microsoft στο CORBA. Ουσιαστικά πρόκειται για την εξέλιξη του COM ώστε να υποστηρίζει κατανεμημένα περιβάλλοντα.

Το COM ορίζει μια binary δομή που λειτουργεί σαν το διαπροσωπείο του αντικειμένου. Ο πελάτης επικοινωνεί με το αντικείμενο μέσω αυτής της δομής. Η περιγραφή γίνεται στην IDL της Microsoft και η υλοποίηση μπορεί να γίνει σε οποιαδήποτε γλώσσα. Ένα αντικείμενο μπορεί να υλοποιεί πολλά διαπροσωπεία και τα αντικείμενα πρέπει να τρέχουν μέσα σε εξυπηρετητές. Με τους όρους του DCOM, το stub του πελάτη ονομάζεται proxy, ενώ του εξυπηρετητή απλώς stub.

Ένα σημαντικό στοιχείο του DCOM είναι ότι ο πελάτης μπορεί να ζητήσει αντικείμενα ενός συγκεκριμένου τύπου, αλλά όχι συγκεκριμένα αντικείμενα. Κάθε φορά που κάνει μια κλήση δημιουργείται ένα νέο αντικείμενο για να τον εξυπηρετήσει. Ένα πλεονέκτημα αυτής της προσέγγισης είναι ότι ο εξυπηρετητής μπορεί να διατηρεί ένα σύνολο αρχικοποιημένων αντικειμένων object pool για να τα διαθέσει αμέσως σε μια κλήση του πελάτη. Αν ο πελάτης επιθυμεί να καλέσει ένα αντικείμενο που βρισκόταν σε συγκεκριμένη κατάσταση μπορεί να χρησιμοποιήσει τα monikers που είναι binary δομές που περιγράφουν πως να αρχικοποιηθεί ένα αντικείμενο ώστε να βρεθεί στην επιθυμητή κατάσταση.

Το DCOM έχει την περίεργη απαίτηση να αναθέτουμε ένα μοναδικό αναγνωριστικό (Globally Unique ID - GUID) σε κάθε διαπροσωπειο (IID) και κάθε κλάση (CLSID). Επίσης, τα αντικείμενα στο COM έχουν πρόσβαση στο λειτουργικό σύστημα, γεγονός

που μπορεί να έχει απρόβλεπτες συνέπειες.

Το DCOM πρακτικά δεν χρησιμοποιείται πια. Από την αρχή πολλοί ήταν διστακτικοί να υιοθετήσουν μια πλατφόρμα που τρέχει μόνο στα λειτουργικά συστήματα της Microsoft. Τελικά, η ίδια η Microsoft προχώρησε παρακάτω, υλοποιώντας νέα πρωτόκολλα (το SOAP αρχικά κι αργότερα νέες τεχνολογίες στο .NET).

4.1.5 Java RMI

Το RMI (Remote Method Invocation)[15] είναι ένας μηχανισμός στη γλώσσα Java που επιτρέπει την κλήση μεθόδων σε απομακρυσμένα αντικείμενα. Είναι κι αυτός ουσιαστικά μια μορφή αντικειμενοστραφούς RPC.

Το RMI ορίζει δυο νέες κατηγορίες κλάσεων, τις Remote και τις Serializable κλάσεις. Οι Remote κλάσεις μπορούν να χρησιμοποιηθούν από απομακρυσμένους χρήστες και οι Serializable κλάσεις μπορούν να μεταφερθούν πάνω από το δίκτυο σε ένα άλλο JVM.

Κάθε Remote κλάση πρέπει να προσδιορίζεται από ένα interface το οποίο πρέπει να είναι public και να επεκτείνει το interface java.rmi.Remote. Επίσης, κάθε μέθοδος πρέπει να δηλώνει ότι πετάει το exception java.rmi.RemoteException γιατί πάντα υπάρχει η πιθανότητα σφάλματος. Ακόμα, κάθε Remote κλάση πρέπει να κληρονομεί την κλάση java.rmi.server.UnicastRemoteObject. Από τη Remote κλάση με τον compiler rmic δημιουργούνται τα stubs και τα skeletons.

Μια Serializable κλάση πρέπει να υλοποιεί απλώς το interface java.io.Serializable. Οι υποκλάσεις μιας Serializable κλάσης είναι κι αυτές Serializable. Οι παράμετροι που περνάνε σε μια Remote κλάση, καθώς και το αποτέλεσμα, πρέπει να είναι Serializable κλάσεις.

Ο πελάτης καλεί το stub, το οποίο αναλαμβάνει να καλέσει τον εξυπηρετητή όπου βρίσκεται η Remote κλάση. Πέρα από τον πελάτη και τον εξυπηρετητή, στο δίκτυο πρέπει να υπάρχει μια object registry. Για την object registry υπάρχει έτοιμο πρόγραμμα, το rmiregisty. Ο εξυπηρετητής καταχωρεί τις κλάσεις του με τη registry και ο πελάτης κάνει αναζητήσεις με βάση τη registry και παίρνει πίσω ένα stub. Αυτά γίνονται μέσω της κλάση java.rmi.Naming, με τις μεθόδους bind και lookup. Όταν ο πελάτης καλεί τη lookup πρέπει να δώσει και το όνομα της registry που επιθυμεί να χρησιμοποιηθεί.

Το RMI είναι σημαντικά πιο απλό στη χρήση του από τις προηγούμενες τεχνολογίες, αφού η υλοποίηση αποκρύπτεται από τον χρήστη. Επίσης, είναι ενσωματωμένο στη γλώσσα Java και δεν απαιτεί την ύπαρξη τρίτων πακέτων. Ωστόσο, παραμένει ένα χαρακτηριστικό που μπορεί να χρησιμοποιηθεί μόνο από τη γλώσσα Java, όταν οι υπόλοιπες τεχνολογίες δεν έχουν τέτοιους περιορισμούς.

4.2 SOAP

4.2.1 Εισαγωγή

Το SOAP (Simple Object Access Protocol) [16] μπορεί να θεωρηθεί το καθιερωμένο πρωτόκολλο για απομακρυσμένες κλήσεις. Αναπτύχθηκε αρχικά από τη Microsoft, αλλά μετά την έκδοση 1.0 ο έλεγχος πέρασε στο W3C [17]. Η τρέχουσα έκδοση είναι η 1.2.

Δύο ήταν τα κύρια συστατικά της επιτυχίας του SOAP: η XML [18] και το HTTP [19]. Κανένα από αυτά τα δυο συστατικά δεν εγγυάται από μόνο του την επιτυχία, αλλά η χρήση τους την κατάλληλη στιγμή επέφερε την επικράτηση του SOAP.

Το SOAP τρέχει πάνω από HTTP και χρησιμοποιεί την XML για την κωδικοποίηση των μηνυμάτων του. Η επιλογή της χρήσης του HTTP[2] σε μια εποχή όπου τα firewalls και τα NATs είναι πολύ συνηθισμένα κρίνεται ιδιαίτερα επιτυχημένη, αφού επέτρεψε την εύκολη λειτουργία του SOAP σε περιβάλλοντα όπου τα υπόλοιπα πρωτόκολλα αποτύγχαναν. Πράγματι, το HTTP είναι μια υπηρεσία η οποία επιτρέπεται πρακτικά παντού. Επίσης, η χρήση της XML, αν και δεν έχει κάποιο εγγενές πλεονέκτημα, έπιασε το πνεύμα της εποχής και τράβηξε αμέσως την προσοχή του κόσμου στο SOAP.

Το SOAP είναι πολύ γενικό πρωτόκολλο και αφήνει μεγάλη ευελιξία στον χρήστη του. Ουσιαστικά προσδιορίζει μονό τη δομή των μηνυμάτων· ακόμα και η χρήση του HTTP είναι προαιρετική και θεωρητικά κάποιος θα μπορούσε να χρησιμοποιήσει ακόμα και UDP. Για τις υπόλοιπες πλευρές του RPC χρησιμποιούνται άλλα πρωτόκολλα. Σαν IDL προτιμάται συνήθως η WSDL (Web Services Description Language) [20] και για την ανακάλυψη υπηρεσιών δημοφιλές είναι το UDDI (Universal Description, Discovery and Integration) [21].

Η έκδοση 1.2 έχει αρκετές διαφορές από τις 1.1 και 1.0. Όσα γράφονται παρακάτω αφορούν την έκδοση 1.2, αν και σε μερικά σημεία αναφέρουμε και τις αλλαγές σε σχέση με τις προηγούμενες εκδόσεις.

4.2.2 Δομή των μηνυμάτων SOAP

Τρία είναι τα βασικά στοιχεία σε ένα SOAP μήνυμα: τα Envelope, Header και Body. Πρέπει να χρησιμοποιούνται με ακριβώς αυτά τα ονόματα και το namespace πρέπει να είναι http://www.w3.org/2003/05/soap-envelope (στις προηγούμενες εκδόσεις το namespace ήταν http://schemas.xmlsoap.org/soap/envelope/). Γενικότερα, το πρότυπο προτείνει τη χρήση namespaces για όλα τα στοιχεία, για να αποφεύγετε η ύπαρξη διφορούμενων στοιχείων.

Η δομή ενός μηνύματος σύμφωνα με όσα περιγράφτηκαν φαίνεται παρακάτω:

```
<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-
    envelope">
```

[2] θεωρητικά μπορεί να χρησιμοποιηθεί και το SMTP

```
 <env:Header>
...
 </env:Header>
 <env:Body>
...
 </env:Body>
</env:Envelope>
```

Το Envelope είναι πάντα το εξωτερικό στοιχείο σε ένα SOAP μήνυμα. Προαιρετικά περιλάμβανει ένα στοιχείο Header και πάντα υπάρχει ακριβώς ένα στοιχείο Body. Το χρήσιμο φορτίο του μηνύματος τοποθετείται στο Body, ενώ στο Header τοποθετούνται συμπληρωματικές πληροφορίες, όπως οδηγίες για την επεξεργασία των στοιχείων του Body.

Το SOAP προβλέπει επίσης το ειδικό στοιχείο Fault που χρησιμοποιείται για την αναφορά σφαλμάτων. Αν το μήνυμα περιέχει Fault τότε αυτό πρέπει να είναι το μοναδικό στοιχείο κάτω από το Body. Παράδειγμα ενός μηνύματος με Fault φαίνεται παρακάτω:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-
    envelope">
  <env:Body>
   <env:Fault>
     <env:Code>
       <env:Value>...</env:Value>
       <env:Subcode>
        <env:Value>...</env:Value>
       </env:Subcode>
     </env:Code>
     <env:Reason>
      <env:Text xml:lang="en-US">...</env:Text>
     </env:Reason>
     <env:Detail>
  ...
      </e:myFaultDetails>
     </env:Detail>
   </env:Fault>
 </env:Body>
</env:Envelope>
```

Το στοιχείο Fault μπορεί να περιέχει τα υποστοιχεία Code, Reason, Node, Role και Detail. Από αυτά μόνο τα δυο πρώτα είναι υποχρεωτικά (πχ στο παράδειγμα τα Node και Role δεν υπήρχαν). Το Code είναι ένας κωδικός που υποδεικνύει το σφάλμα. Μπορεί να πάρει μόνο τις τιμές που φαίνονται στον Πίνακα 4.2.2, αλλά

Όνομα	Σημασία
VersionMismatch	Η έκδοση του Envelope δεν ήταν αυτή που αναμενόταν.
MustUnderstand	Ο δέκτης δεν μπορούσε να επεξεργασθεί ένα στοιχείο του Header που ήταν σημειωμένο ως mustUnderstand.
DataEncodingUnknown	Ο δέκτης δεν μπορούσε να αναγνωρίσει την κωδικοποίηση των δεδομένων.
Sender	Η δομή του μηνύματος δεν ήταν σωστή. Υποδεικνύει ότι δεν πρέπει να ξανασταλεί το μήνυμα αν δεν γίνουν αλλαγές.
Receiver	Υποδεικνύει ότι ο δέκτης δεν μπορούσε να επεξεργαστεί το μήνυμα, αν και αυτό ήταν σε κατάλληλη μορφή.

Πίνακας 4.1: Κώδικες σφαλμάτων SOAP

το Subcode μπορεί να πάρει οποιαδήποτε τιμή για να διαφοροποιηθούν οι επιμέρους περιπτώσεις. Το Reason πρέπει να περιέχει κείμενο που επεξηγεί καλύτερα το σφάλμα. Το κάθε στοιχείο πρέπει να προσδιορίζει το attribute xml:lang έτσι ώστε να είναι δυνατόν να δωθούν εξηγήσεις σε πολλές γλώσσες. Επίσης, το Detail μπορεί να χρησιμοποιηθεί από την εφαρμογή για να τοποθετήσει επιπλέον πληροφορίες σχετικές με το σφάλμα. Ο ρόλος των Node και Role δεν θα εξηγηθεί εδώ.

4.2.3 SOAP-RPC

Μέχρι τώρα είδαμε μόνο τη δομή των μηνυμάτων SOAP. Αυτά τα μηνύματα μπορούν να χρησιμοποιηθούν για οποιοδήποτε σκοπό, αλλά η πιο συνηθισμένη χρήση είναι το SOAP-RPC. Σε αυτή την περίπτωση ακολουθείται το μοντέλο πελάτη - εξυπηρετητή: ο πελάτης στέλνει ένα request και ο εξυπηρετητής απαντάει με ένα response. Αυτά τα μηνύματα κωδικοποιούνται σύμφωνα με το SOAP.

Το μήνυμα μιας RPC κλήσης περιέχει το όνομα της μεθόδου και τις παραμέτρους. Κάτω από το Body τοποθετείται ένα στοιχείο με το όνομα της μεθόδου και κάτω από αυτό μπαίνουν οι παράμετροι. Ένα παράδειγμα από το [22] φαίνεται παρακάτω:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap–
    envelope" >
 <env:Header>
   <t:transaction
```

```
            xmlns:t="http://thirdparty.example.org/
                transaction"
            env:encodingStyle="http://example.com/encoding"
            env:mustUnderstand="true" >5</t:transaction>
 </env:Header>
 <env:Body>
  <m:chargeReservation
      env:encodingStyle="http://www.w3.org/2003/05/soap-
          encoding"
        xmlns:m="http://travelcompany.example.org/">
   <m:reservation xmlns:m="http://travelcompany.example.org
       /reservation">
    <m:code>FT35ZBQ</m:code>
   </m:reservation>
   <o:creditCard xmlns:o="http://mycompany.example.com/
       financial">
    <n:name xmlns:n="http://mycompany.example.com/employees
        ">
            John Gaule
    </n:name>
    <o:number>123456789099999</o:number>
    <o:expiration>2005-02</o:expiration>
   </o:creditCard>
  </m:chargeReservation>
 </env:Body>
</env:Envelope>
```

Στο παράδειγμα η μέθοδος που καλείται είναι η chargeReservation και οι παράμετροι είναι οι reservation και creditCard. Η παράμετρος creditCard είναι σύνθετη δομή και αποτελείται από τα name, number και expiration. Οι διάφοροι τύποι των δεδομένων θα συζητηθούν στη συνέχεια. Μπορούμε να φανταστούμε ότι στη γλώσσα Java η κλήση θα ήταν κάπως έτσι:

<object>.chargeReservation(Reservation res, CreditCard card);

Φυσικά γεννάται το ερώτημα πού ορίζεται το <object>. Στις προηγούμενες εκδόσεις του SOAP υπήρχε μια αοριστία όσον αφορά το θέμα αυτό, αλλά η έκδοση 1.2 ορίζει ότι το αντικείμενο προσδιορίζεται από το URI [9] όπου γίνεται το request. Δηλαδή, αυτή η πληροφορία δεν περιέχεται στο μήνυμα SOAP αλλά στα χαμηλότερα πρωτόκολλα που μεταφέρουν το μήνυμα.

Οι παράμετροι που κωδικοποιούνται με το SOAP μπορούν να είναι είτε σύνθετοι τύποι ή απλοί. Οι απλοί, αλφαριθμητικοί τύποι γράφονται απευθίας όπως είναι. Οι σύνθετοι τύποι μπορεί να είναι είτε struct ή array. Αν θέλουμε να τους διαχωρίσουμε μπορούμε να χρησιμοποιήσουμε το attribute nodeType. Σε ένα array η σειρά των

στοιχείων έχει σημασία, ενώ σε ένα struct τα στοιχεία ξεχωρίζονται με βάση το όνομά τους. Ένα array μπορεί να έχει τα επιπλέον attributes itemType και arraySize, που ορίζουν τον τύπο και το πλήθος των στοιχείων αντίστοιχα. Σε κάθε περίπτωση, αν έχουμε κάπου ορισμένο ένα schema, μπορούμε να το χρησιμοποιήσουμε βάζοντας το κατάλληλο namespace και το attribute xsi:type. Έτσι, όλοι οι τύποι που ορίζει το XML Schema [23] είναι άμεσα διαθέσιμοι.

Το μήνυμα που περιέχει την απάντηση του εξυπηρετητή έχει παρόμοια δομή με την προηγούμενη. Κατά σύμβαση, το κύριο στοιχείο στο Body της απάντησης έχει το όνομα της μεθόδου που καλέσθηκε, προσθέτοντας ένα 'Response' στο τέλος. Μια πιθανή απάντηση στο προηγούμενο παράδειγμα φαίνεται παρακάτω:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-
    envelope" >
 <env:Header>
     <t:transaction
        xmlns:t="http://thirdparty.example.org/transaction"
          env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true">5</t:transaction>
 </env:Header>
 <env:Body>
     <m:chargeReservationResponse
          env:encodingStyle="http://www.w3.org/2003/05/soap-
              encoding"
               xmlns:m="http://travelcompany.example.org/">
        <m:code>FT35ZBQ</m:code>
        <m:viewAt>
          http://travelcompany.example.org/reservations?code
              =FT35ZBQ
        </m:viewAt>
     </m:chargeReservationResponse>
 </env:Body>
</env:Envelope>
```

Παρατηρούμε ότι το κύριο στοιχείο ονομάζεται chargeReservationResponse, υποδεικνύοντας ότι αποτελεί απάντηση στην κλήση της μεθόδου chargeReservation. Επίσης, βλέπουμε ότι το αποτέλεσμα μπορεί να περιλαμβάνει κι αυτό πολλά στοιχεία (code και viewAt).

Φυσικά, υπάρχει η περίπτωση σφάλματος, κατά την οποία η απάντηση θα ήταν κάπως έτσι:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-
    envelope"
              xmlns:rpc='http://www.w3.org/2003/05/soap-rpc'>
```

```
  <env:Body>
   <env:Fault>
     <env:Code>
       <env:Value>env:Sender</env:Value>
       <env:Subcode>
        <env:Value>rpc:BadArguments</env:Value>
       </env:Subcode>
     </env:Code>
     <env:Reason>
      <env:Text xml:lang="en-US">Processing error</env:Text
          >
     </env:Reason>
     <env:Detail>
      <e:myFaultDetails
        xmlns:e="http://travelcompany.example.org/faults">
        <e:message>Name does not match card number</e:
            message>
        <e:errorcode>999</e:errorcode>
      </e:myFaultDetails>
     </env:Detail>
   </env:Fault>
 </env:Body>
</env:Envelope>
```

Το μήνυμα σφάλματος είναι παρόμοιο με αυτό που δείξαμε προηγουμένως. Παρατηρούμε ότι το Subcode έχει τιμή rpc:BadArguments, που σημαίνει ότι ο δέκτης δεν μπορούσε να αναλύσει συντακτικά τις παραμέτρους. Ένα άλλο προκαθορισμένο SubCode είναι το rpc:ProcedureNotPresent, που σημαίνει ότι δεν υπάρχει η συνάρτηση που ζητήθηκε.

4.2.4 HTTP Binding

Όπως είπαμε, το SOAP δεν επιβάλλει τη χρήση κάποιου συγκεκριμένου πρωτοκόλλου χαμηλότερης τάξης. Σχεδόν αποκλειστικά όμως χρησιμοποιείται σε συνδυασμό με το HTTP και το πρωτόκολλο περιγράφει αυτή τη χρήση.

Το HTTP έχει κι αυτό μηνύματα Request και Response και υπάρχει μια προφανής αντιστοίχιση με αυτά του SOAP. Σε ένα Request μπορεί να χρησιμοποιηθεί είτε η μέθοδος POST ή η GET. Βέβαια, η χρήση της GET επιβάλλει σημαντικούς περιορισμός και ουσιαστικά δεν στέλνουμε, αλλά μόνο λαμβάνουμε SOAP μήνυμα. Συνήθως χρησιμοποιείται η POST και το SOAP μήνυμα τοποθετείται αυτούσιο στο body του HTTP. Στα headers του HTTP πρέπει να δηλωθεί το Content-Type ως application/soap+xml (παλιότερα ήταν text/xml. Παλιότερα χρησιμοποιούταν ένα επιπλέον header, το SOAPAction, αλλά αυτό έχει τώρα καταργηθεί (προαιρετικά μπορεί να δηλωθεί με το attribute action μετά το application/soap+xml).

Στην απάντηση από τον εξυπηρετητή το SOAP μήνυμα τοποθετείται και πάλι αυτούσιο στο body του HTTP. Επίσης, τα HTTP Return Codes χρησιμοποιούνται για να υποδείξουν την επιτυχία ή την αποτυχία της κλήσης. Το 200 και γενικότερα return codes της μορφής 2xx σημαίνουν επιτυχία και 4xx, 5xx σημαίνουν αποτυχία. Από τον Πίνακα 4.1 το env:Sender αντιστοιχεί στο 400, ενώ όλα τα υπόλοιπα στο 500.

4.2.5 WSDL

Όπως έχουμε ήδη αναφέρει, το SOAP δεν καθορίζει κάποια IDL για την περιγραφή του διαπροσωπείου των υπηρεσιών. Πολύ συχνά για αυτό τον σκοπό χρησιμοποιείται ένα άλλο πρωτόκολλο, το WSDL (Web Services Description Language). Λόγω της διαδεδομένης χρήσης του, στη συνέχεια θα περιγράψουμε σύντομα τη δομή του. Η τρέχουσα έκδοση του wsdl είναι η 1.1, η οποία έχει υποβληθεί στο W3C αλλά δεν έχει υιοθετηθεί σαν πρότυπο. Το W3C εργάζεται προς την προτυποποίηση της έκδοσης 2.0, η οποία δεν έχει οριστικοποιηθεί ακόμα.

Ένα wsdl έγγραφο είναι ένα αρχείο xml που περιγραφεί τον τρόπο χρήσης ενός web service. Περιέχει το είδος και τον αριθμό των παραμέτρων, καθώς και τα μηνύματα που ανταλλάσσονται. Το root στοιχείο σε ένα wsdl αρχείο ονομάζεται definitions και μπορεί να έχει τα εξής υποστοιχεία:

types Περιγράφει τους τύπους των δεδομένων που χρησιμοποιούνται.

message Αντιπροσωπεύει μια αφηρημένη αναπαράσταση των δεδομένων που ανταλλάσσονται.

portType Ένα σύνολο απο αφηρημένες λειτουργίες. Κάθε λειτουργία προσδιορίζει ένα εισερχόμενο κι ένα εξερχόμενο μήνυμα.

binding

port Προσδιορίζει τη διεύθυνση για ένα binding.

service Χρησιμοποιείται για να ομαδοποιήσει ένα σύνολο από ports

Παρακάτω βλέπουμε ένα σχετικά σύνθετο wsdl αρχείο, βασιζόμενοι στο οποίο θα αναλύσουμε τις βασικές έννοιες.

```
<?xml version="1.0"?>
<definitions name="StockQuote"
          targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
<message name="GetTradePriceInput">
    <part name="tickerSymbol" element="xsd:string"/>
    <part name="time" element="xsd:timeInstant"/>
</message>

<message name="GetTradePriceOutput">
    <part name="result" type="xsd:float"/>
</message>

<portType name="StockQuotePortType">
    <operation name="GetTradePrice">
       <input message="tns:GetTradePriceInput"/>
       <output message="tns:GetTradePriceOutput"/>
    </operation>
</portType>

<binding name="StockQuoteSoapBinding" type="tns:
   StockQuotePortType">
    <soap:binding style="rpc" transport="http://schemas.
       xmlsoap.org/soap/http"/>
    <operation name="GetTradePrice">
       <soap:operation soapAction="http://example.com/
          GetTradePrice"/>
       <input>
           <soap:body use="encoded" namespace="http://example
              .com/stockquote"
                     encodingStyle="http://schemas.xmlsoap.
                        org/soap/encoding/"/>
       </input>
       <output>
           <soap:body use="encoded" namespace="http://example
              .com/stockquote"
                     encodingStyle="http://schemas.xmlsoap.
                        org/soap/encoding/"/>
       </output>
    </operation>>
</binding>

<service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:
       StockQuoteBinding">
       <soap:address location="http://example.com/stockquote"
```

```
            />
        </port>
    </service>
</definitions>
```

Καταρχάς βλέπουμε ότι ορίζονται δύο μηνύματα, τα GetTradePriceInput και GetTradePriceOutput. Το πρώτο περιέχει δύο παραμέτρους, ένα αλφαριθμητικό και μια χρονική στιγμή και το δεύτερο μια παράμετρο, έναν δεκαδικό αριθμό.

Από το portType StockQuotePortType βλέπουμε ότι η λειτουργία GetTradePrice έχει ως εισερχόμενο μήνυμα το GetTradePriceInput κι εξερχόμενο GetTradePriceOutput. Με τις πληροφορίες που συλλέξαμε ως εδώ συμπεραίνουμε ότι η δήλωση της συνάρτησης είναι κάπως έτσι:

float GetTradePrice(string, timeInstant)

Το binding προσφέρει πληροφορίες για τις SOAP κλήσεις προς αυτό το web service. Ορίζει ότι πρέπει να χρησιμοποιηθούν SOAP-RPC μηνύματα, με συγκεκριμένο namespace. Παρατηρούμε ότι ορίζει και το SOAPAction που δεν υποστηρίζεται πια στο SOAP 1.2. Αντίθετα, η πληροφορία για το URI περιέχεται στο port, που βρίσκεται κάτω από το service.

Το στοιχείο types μπορεί να χρησιμοποιηθεί για να περιγράψει περίπλοκους τύπους δεδομένων. Μέσα περιλαμβάνεται η περιγραφή των τύπων και για αυτό τον σκοπό συστήνεται η χρήση του XML Schema [23]. Ένα παράδειγμα φαίνεται παρακάτω:

```
<?xml version="1.0"?>
<definitions name="StockQuote"
          targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
...
  <types>
      <!-- all type declarations are in a chunk of xsd -->
    <xsd:schema targetNamespace="http://example.com/stockquote.
        xsd"
            xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

      <xsd:element name="TradePriceRequest">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="tickerSymbol" type="string"/>
          </xsd:all>
```

```
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="TradePrice">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="price" type="float"/>
          </xsd:all>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
...
</definitions>
```

Σε αυτό το παράδειγμα, το types περιέχει ένα XML Schema που περιγράφει δύο τύπους δεδομένων, τα TradePriceRequest και TradePrice. Αυτοί οι τύποι μπορούν να χρησιμοποιηθούν στη συνέχεια του wsdl αρχείου, αρκεί να χρησιμοποιήσουμε το πρόθεμα xsd1.

Βιβλιογραφία

[1] "A High-Level Framework for Network-Based Resource Sharing" RFC 707, Internet Engineering Task Force, Jan. 1976

[2] Sun Microsystems, Inc. "XDR: External Data Representation Standard" RFC 1014, Internet Engineering Task Force, Jun. 1987

[3] Sun Microsystems, Inc. "RPC: Remote Procedure Call Protocol Specification Version 2" RFC 1057, Internet Engineering Task Force, Jun. 1988

[4] Sun Microsystems, Inc. "NFS: Network File System Protocol Specification" RFC 1094, Internet Engineering Task Force, Mar. 1989

[5] B. Callaghan, B. Pawlowski, P. Staubach "NFS Version 3 Protocol Specification" RFC 1813, Internet Engineering Task Force, Jun. 1995

[6] R. Srinivasan "RPC: Remote Procedure Call Protocol Specification Version 2" RFC 1831, Internet Engineering Task Force, Aug. 1995

[7] R. Srinivasan "XDR: External Data Representation Standard" RFC 1832, Internet Engineering Task Force, Aug. 1995

[8] R. Srinivasan "Binding Protocols for ONC RPC Version 2" RFC 1833, Internet Engineering Task Force, Aug. 1995

[9] T. Berners-Lee, R. Fielding, L. Masinter "Uniform Resource Identifiers (URI): Generic Syntax" RFC 2396, Internet Engineering Task Force, Aug. 1998

[10] "Common Object Request Broker Architecture" http://www.corba.org

[11] "Object Management Group" http://www.omg.org

[12] "Unified Modeling Language" http://www.uml.org

[13] Michi Henning "The Rise and Fall of CORBA" ACM Queue vol. 4, no. 5 - June 2006

[14] Microsoft Corp. "COM: Component Object Model Technologies" http://www.microsoft.com/com/default.mspx

[15] Sun Microsystems "Java Remote Method Invocation (Java RMI)" http://java.sun.com/products/jdk/rmi/

[16] W3C "SOAP Version 1.2" http://www.w3.org/TR/soap12

[17] W3C "World Wide Web Consortium" http://www.w3.org/

[18] W3C "Extensible Markup Language" http://www.w3.org/XML/

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee "Hypertext Transfer Protocol – HTTP/1.1" RFC 2616, Internet Engineering Task Force, Jun. 1999

[20] W3C "Web Services Description Language" http://www.w3.org/TR/wsdl

[21] "Universal Description, Discovery and Integration (UDDI)" http://www.uddi.org

[22] W3C "SOAP Version 1.2 Part 0: Primer" http://www.w3.org/TR/2003/REC-soap12-part0-20030624/

[23] W3C "XML Schema" http://www.w3.org/XML/Schema

Κεφάλαιο 5

Υλοποίηση

Σκοπός μας ήταν η ανάπτυξη συστήματος ανακάλυψης κι επικοινωνίας υπηρεσιών σε περιβάλλον τοπικού δικτύου. Επίσης, ανάμεσα στους σκοπούς μας ήταν και η χρήση μιας πλατφόρμας όπου είναι δυνατόν να διαχειριστούμε τις υπηρεσίες δυναμικά. Όπως αναφέρθηκε, σαν πλατφόρμα υπηρεσιών επιλέξαμε το OSGi και το επεκτείναμε έτσι ώστε να μπορεί να χρησιμοποιήσει απομακρυσμένες υπηρεσίες. Για την ανακάλυψη διαλέξαμε το SLP και για τις κλήσεις το SOAP. Το τελικό αποτέλεσμα το ονομάζουμε "mediator" και μπορεί να εγκατασταθεί και να χρησιμοποιηθεί με ευκολία πάνω στην πλατφόρμα OSGi.

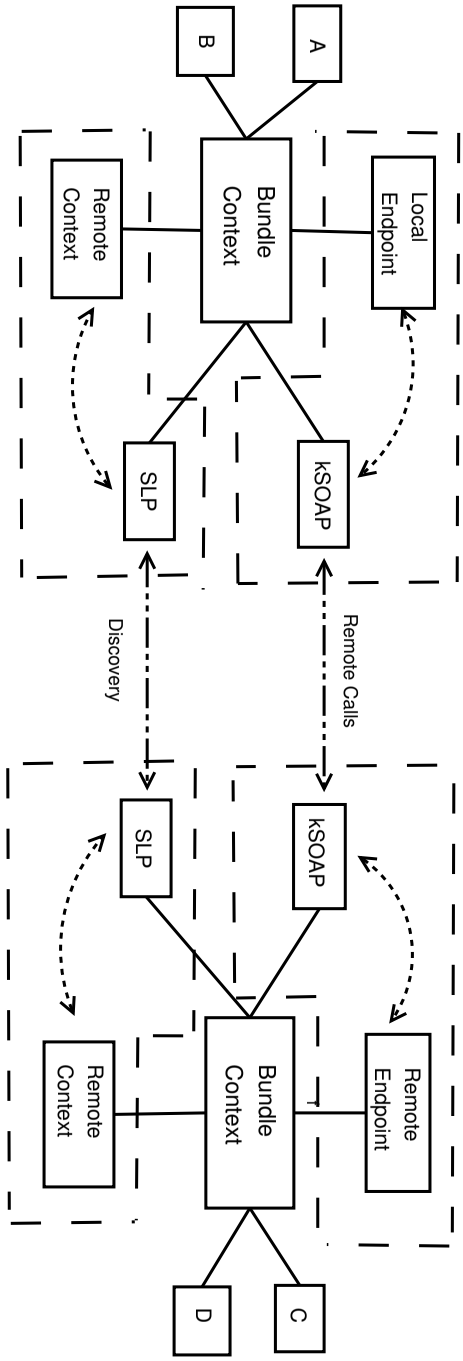
Ακολουθεί η περιγραφή της υλοποίησης του mediator. Σημειώνουμε ότι σε αυτό το Κεφάλαιο χρησιμοποιούμε χωρίς ιδιαίτερες εξηγήσεις πολλά από τα στοιχεία που παρουσιάσαμε στα προηγούμενα Κεφάλαια.

5.1 Αρχιτεκτονική του Mediator

Πιστεύουμε ότι πριν εισέλθουμε στις λεπτομέρειες της υλοποίησης πρέπει να έχουμε μια γενική εικόνα του mediator. Μια κατατοπιστική απεικόνιση φαίνεται στο Σχήμα 5.1.

Τα jSLP και kSOAP είναι τρίτα πακέτα που εγκαθίστανται εύκολα στο OSGi framework. Ο mediator δηλώνει δύο υπηρεσίες, τις RemoteContext και LocalEndpoint κι ένα web service, το RemoteEndpoint. Το RemoteContext ασχολείται με την ανακάλυψη των υπηρεσιών, ενώ τα δύο endpoints υλοποιούν τις απομακρυσμένες κλήσεις υπηρεσιών.

Ένα bundle που θέλει να κάνει μια υπηρεσία του διαθέσιμη σε απομακρυσμένα OSGi frameworks πρέπει να δηλώσει την υπηρεσία του στο RemoteContext, ακριβώς όπως θα τη δήλωνε στο BundleContext για να είναι διαθέσιμη τοπικά. Επίσης, αυτός που επιθυμεί να ανακαλύψει μια απομακρυσμένη υπηρεσία πρέπει να δηλώσει έναν RemoteServiceListener στο RemoteContext για να ειδοποιηθεί όταν βρεθεί η υπηρεσία. Το RemoteContext συνεργάζεται με το jSLP ώστε να πραγματοποιηθεί η ανακάλυψη των υπηρεσιών. Δηλαδή, ο χρήστης επικοινωνεί με το RemoteContext



Σχήμα 5.1: Αρχιτεκτονική του mediator

κι αυτό αναλαμβάνει τη ρύθμιση του jSLP.

Όταν ανακαλυφθεί μια υπηρεσία ο ενδιαφερόμενος παίρνει ένα ServiceURL που δείχνει σε αυτή. Αν θέλει να καλέσει την υπηρεσία, δίνει το ServiceURL στο LocalEndpoint, μαζί με τη μέθοδο που τον ενδιαφέρει και τις παραμέτρους της κλήσης. Το LocalEndpoint δημιουργεί μια SOAP κλήση στο RemoteEndpoint, που είναι ένα web service το οποίο εξυπηρετεί όλες τις διαθέσιμες υπηρεσίες. Το RemoteEndpoint μπορεί να καλέσει μόνο υπηρεσίες που έχουν καταχωρηθεί στο RemoteContext. Η πραγματοποίηση των SOAP κλήσεων γίνεται με το πακέτο kSOAP.

5.2 Knopflerfish

Το Knopflerfish είναι μια υλοποίηση ανοικτού κώδικα του OSGi framework που διατίθεται με BSD άδεια. Η έκδοση 2.0.0 είναι πλήρως συμβατή με το Release 4 και παρέχει μερικά επιπλέον bundles. Στη συνέχεια θα αναλύσουμε σύντομα τη λειτουργία του Knopflerfish.

5.2.1 Οργάνωση των αρχείων

Αν εγκαταστήσουμε το Knopflerfish στον φάκελο KF_HOME τότε κάτω από το KF_HOME/knopferfish.org/ θα δούμε τους παρακάτω φακέλους:

ant Εδώ περιέχονται μερικά αρχεία που διευκολύνουν τη χρήση του ant για τη δημιουργία κάποιου bundle.

htdocs Εδώ περιέχεται αντίγραφο της html σελίδας του knopflerfish, μαζί με όλη την τεκμηρίωση.

osgi/bundles Εδώ τοποθετείται ο πηγαίος κώδικας των bundles. Κάθε bundle τοποθετείται σε δικό του φάκελο.

osgi/framework Εδώ περιέχεται ο πηγαίος κώδικας του ίδιου του Knopflerfish.

osgi/jars Εδώ τοποθετούνται τα jars των bundles. Το κάθε bundle έχει το δικό του φάκελο.

Υπάρχουν διάφορες διανομές του Knopflerfish διαθέσιμες στο Internet. Η πλήρης διανομή περιέχει όλα τα παραπάνω και μερικά ακόμα αρχεία και είναι πιο κατάλληλη αν μας ενδιαφέρει η ανάπτυξη εφαρμογών. Αν το Knopflerfish θα τρέχει μόνο έτοιμα προγράμματα, μπορούμε να κατεβάσουμε την απλή διανομή, η οποία περιέχει μόνο το φάκελο με τα jars. Ενδεικτικά αναφέρουμε ότι η πλήρης διανομή καταλαμβάνει περίπου 20 MB, ενώ η απλή περίπου 6 MB.

5.2.2 Ανάπτυξη εφαρμογών στο Knopflerfish

Για να χρησιμοποιήσουμε ένα έτοιμο bundle αρκεί να το τοποθετήσουμε το jar στο δικό του φάκελο κάτω από το osgi/jars και να το εγκαταστήσουμε (λεπτομέρεις παρακάτω). Σημειώνουμε ότι οι τοποθεσίες που αναφέρουμε δεν είναι υποχρεωτικές, αλλά βοηθούν στη διατήρηση μιας λογικής οργάνωσης.

Αν θέλουμε να δημιουργήσουμε δικό μας bundle, φτιάχνουμε ένα νέο φάκελο κάτω από το osgi/bundles. Μέσα δημιουργούμε τον φάκελο src, όπου τοποθετούμε τον πηγαίο κώδικα. Στο φάκελο resources μπορούμε να βάλουμε αρχεία που θέλουμε να περιληφθούν στο jar, ενώ στο φάκελο lib μπορούμε να βάλουμε τρίτες βιβλιοθήκες που χρησιμοποιούμε.

Αντιγράφουμε από τον φάκελο ant το αρχείο build_examle.xml και το μετονομάζουμε σε build.xml. Αυτό το αρχείο κάνει include το bundlebuild_include.xml και περιέχει μόνο τις τροποποιήσεις που θέλουμε. Ένα παράδειγμα φαίνεται παρακάτω:

```
<?xml version="1.0"?>

<!DOCTYPE project [
  <!ENTITY bundlebuild_include  SYSTEM "../../../ant/
     bundlebuild_include.xml">
]>

<project name="example" default="all">

  <property name="topdir"        location="../.."/>

  <property name="impl.pattern"
            value="org/acme/impl/**"/>

  <property name="api.pattern"
            value="org/acme/**"/>

  <path id="bundle.compile.path">
    <pathelement location="${topdir}/jars/test/test.jar"/>
  </path>

  &bundlebuild_include;

</project>
```

Με το παραπάνω αρχείο ορίζουμε ότι το όνομα του bunlde μας είναι example, τα interfaces των υπηρεσιών του βρίσκονται στο φάκελο org/acme και η υλοποίηση στο φάκελο org/acme/impl. Επίσης, ορίζουμε ότι εισάγονται κλάσεις από το test.jar. Υπάρχουν επιπλέον επιλογές για τον ορισμό έκδοσης, δημιουργού, κλπ, αλλά δεν τις παρουσιάζουμε εδώ.

Όταν βρισκόμαστε στο φάκελο του bundle μας τρέχοντας απλώς ant θα δημιουργηθεί το jar και θα τοποθετηθεί στο jars/<name>/<name>.jar.

Σημειώνουμε ότι υπάρχει και plugin για το Eclipse για την ανάπτυξη εφαρμογών στο Knopflerfish, αλλά δεν το χρησιμοποιήσαμε.

5.2.3 Χρήση του Knopflerfish

Για να εκκινήσουμε το knopflerfish πηγαίνουμε στο φάκελο osgi και τρέχουμε java -jar framework.jar. Το Knopflerfish εκτός από τα bundles που προβλέπει το OSGi εκκινεί και μερικά ακόμη τα οποία θεωρεί χρήσιμα. Για παράδειγμα, η διαχείριση μπορεί να γίνει είτε από τη γραμμή εντολών ή από το γραφικό περιβάλλον που έχει (KF Desktop). Αν δεν υπάρχει κάποια οθόνη (πχ σε μια ενσωματωμένη συσκευή), τότε πρέπει να σταματήσουμε το KF Desktop, το οποίο συνήθως είναι το bundle 17. Ένα άλλο bundle που ίσως θέλουμε να σταματήσουμε είναι το telnet. Τα bundles που θα εγκατασταθούν κι εκκινηθούν την πρώτη φορά περιέχονται στο αρχείο init.xargs.

Η διαχείριση από τη γραμμή εντολών είναι πολύ απλή. Παρακάτω φαίνονται οι βασικές λειτουγίες (υποθέτοντας ότι έχουμε βάλει ένα bundle στο δικό του φάκελο κάτω από το osgi/jars).

```
> install file:jars/<name>/<name>.jar
Installed: <name> (#31)
> start 31
Started: <name> (#31)
> stop 31
Stopped: <name> (#31)
```

Φυσικά όλες οι παραπάνω λειτουργίες μπορούν να γίνουν και από το γραφικό περιβάλλον. Μάλιστα, το KF Desktop προσφέρει επιπλέον πληροφορίες: για κάθε bundle μπορούμε να δούμε το Manifest του, τις υπηρεσίες που εισάγει κι εξάγει και το πιο σημαντικό, το log του. Το τελευταίο είναι διαθέσιμο κι από τη γραμμή εντολών γράφοντας

> log show 31

Πιθανότατα βέβαια το γραφικό περιβάλλον είναι πιο φιλικό για τους περισσότερους.

5.3 SLP

5.3.1 Σχεδιαστικές Αποφάσεις

Στο SLP οι αναζητήσεις γίνονται με βάση το ServiceType, που υπενθυμίζουμε ότι έχει τη μορφή:

service:<abstract-type>:<concrete-type>

Ζητούμενο είναι να γίνεται αναζήτηση με βάση το όνομα ενός interface, οπότε τελικά αποφασίστηκε να χρησιμοποιούμε την παρακάτω μορφή ServiceType:

service:osgi:<interface-name>

Δηλαδή, όλες οι υπηρεσίες έχουν αφηρημένο τύπο "osgi" και συγκεκριμένο τύπο το όνομα του interface που υλοποιούν. Μάλιστα, στο όνομα αντικαθιστούμε το σύμβολο '.' με το '_' για συμβατότητα με το SLP.

Μια σχετικά προφανής απόφαση ήταν να αντιστοιχηθούν τα properties μιας υπηρεσίας στο OSGi με τα attributes στο SLP. Και τα δύο είναι ζεύγη key=value οπότε υπάρχει πλήρης συμβατότητα μεταξύ τους.

5.3.2 jSLP

Εγκατάσταση και ρύθμιση

Το jSLP είναι μια υλοποίηση του πρωτοκόλλου SLP στη γλώσσα Java. Αναπτύχθηκε από το εργαστήριο Information and Communication Systems Research Group (IKS) του πανεπιστημίου ETHZ και είναι ελεύθερα διαθέσιμο. Υπάρχουν δύο εκδόσεις, μια που διανείμεται ως OSGi bundle και μια που είναι αυτοτελής εφαρμογή. Προφανώς εμείς χρησιμοποιήσαμε το OSGi bundle και πιο συγκεκριμένα την έκδοση 0.5.6.

Το jSLP φιλοξενείται από το SourceForge κι από εκεί κατεβάσαμε το jslp-osgi-0.5.6.jar (που είναι πολύ μικρό - περίπου 56 Kbytes. Η εγκατάσταση γίνεται με τα εξής απλά βήματα:

1. Τοποθέτηση του αρχείου στο φάκελο osgi/jars/jslp/.
2. Εκκίνηση του Knopflerfish (αν δεν λειτουργεί ήδη).
3. Εκτέλεση της εντολής: install file:jars/jslp/jslp-osgi-0.5.6.jar

Μετά από αυτά τα βήματα θα εμφανιστεί στο KF Desktop ένα εικονίδιο που αντιπροσωπεύει το bundle του jSLP. Αν θέλουμε μπορούμε να το εκκινήσουμε για να το δοκιμάσουμε.

Σημειώνουμε ότι ίσως εμφανιστούν κάποια προβλήματα. Καταρχάς, το πρωτόκολλο SLP ακούει στη θύρα 427. Σε πολλά λειτουργικά συστήματα μόνο χρήστες με ειδικά προνόμια μπορούν να ακούν σε θύρες μικρότερες από το 1024. Στα Windows αυτό δεν υπάρχει τέτοιος περιορισμός, αλλά σε Linux, *BSD και γενικότερα UNIX-οειδή συστήματα αυτό είναι πρόβλημα. Μια αφελής προσέγγιση θα ήταν να τρέχει το framework ένας διαχειριστής, αλλά αυτό θέτει ζητήματα ασφαλείας, αφού μέσα στο framework μπορεί να τρέχουν πολλά προγράμματα και θα ήταν δύσκολο να καθοριστεί ποια από αυτά είναι ασφαλή. Μια λύση θα ήταν να τρέχει το Knopflerfish

κάποιος διαχειριστής, αλλά μέσα σε ένα virtual machine έτσι ώστε να περιορίζονται οι πιθανές παρενέργειες. Η πιο σωστή προσέγγιση θα ήταν να χρησιμοποιούνται οι δυνατότητες που προσφέρει το κάθε σύστημα για τέτοιες περιπτώσεις. Για παράδειγμα, σε OpenBSD και NetBSD το εργαλείο systrace μπορεί να επιτρέψει σε προγράμματα που εκτελούνται από απλούς χρήστες να πραγματοποιήσουν συγκεκριμένα system calls. Στην περίπτωση μας, το system call που μας ενδιαφέρει είναι το bind.

Επίσης, προβλήματα μπορούν να προκύψουν σε συστήματα που έχουν πολλαπλά network interfaces, όπως για παράδειγμα ένας δρομολογητής που έχει πολλές κάρτες δικτύου ή ένα απλό μηχάνημα που συνδέεται σε διάφορα virtual LANs. Σε αυτή την περίπτωση, το jSLP θα διαλέξει ένα interface στην τύχη και θα ακούει σε αυτό. Είναι αυτονόητο ότι αυτή δεν είναι πάντα η επιθυμητή συμπεριφορά. Για παράδειγμα, μπορεί το jSLP να διαλέξει να ακούει σε ένα VLAN αντί για το τοπικό δίκτυο και μάλλον δεν θέλουμε να χρησιμοποιούμε το SLP πάνω από το διαδίκτυο. Ευτυχώς, υπάρχει η δυνατότητα ρύθμισης αυτής της επιλογής. Το jSLP διαβάζει ένα αρχείο, το jslp.properties, στο οποίο μπορούμε να ορίσουμε διάφορες ιδιότητες. Στη συγκεκριμένη περίπτωση γράφουμε για παράδειγμα:

```
net.slp.interfaces=192.168.5.1
```

με αποτέλεσμα το jSLP να ακούει στη διεύθυνση 192.168.5.1. Αν θέλουμε, μπορούμε να προσθέσουμε κι άλλες διευθύνσεις, χωρίζοντάς τις με κόμμα. Το σύνολο των επιλογών που μπορούμε να βάλουμε στο αρχείο jslp.properties φαίνονται στον Πίνακα 5.3.2. (Σημειώνουμε ότι το αρχείο jslp.properties τοποθετείται στο φάκελο osgi/).

5.3.3 Χρήση του jSLP

Καταρχάς, σημειώνουμε ότι το διαπροσωπείο του jSLP ακολουθεί τις οδηγίες του RFC 2614.

Το jSLP καταχωρεί δυο υπηρεσίες στο OSGi: τον Locator και τον Advertiser. Ο Locator είναι υπεύθυνος για την ανακάλυψη των υπηρεσιών που μας ενδιαφέρουν, ενώ ο Advertiser διαφημίζει τις υπηρεσίες μας στο δίκτυο. Δύο άλλες σημαντικές κλάσεις είναι οι ServiceURL και ServiceType, των οποίων οι έννοιες ορίστηκαν στο κεφάλαιο του SLP. Αναφέρουμε σύντομα ότι το ServiceType δηλώνει το είδος της υπηρεσίας και το ServiceURL περιέχει το ServiceType μαζί με τη διεύθυνση όπου είναι διαθέσιμη η υπηρεσία. Όλες αυτές οι κλάσεις ανήκουν στο πακέτο ch.ethz.iks.slp.

Ένα αντικείμενο της κλάσης ServiceType δημιουργείται δίνοντας του το string που αντιπροσωπεύει τον τύπο της υπηρεσίας. Το ServiceURL περιέχει ένα επιπλέον στοιχείο, το lifetime. Όταν ένα ServiceURL καταχωρείται στον Advertiser το lifetime δείχνει πόση ώρα προβλέπεται να είναι διαθέσιμη η υπηρεσία. Ο Advertiser παρέχει αυτή την πληροφορία σε όσους δώσει το ServiceURL, έτσι ώστε να μπορούν να κάνουν κι αυτοί τις εκτιμήσεις του. Όταν το lifetime λήξει, ο Advertiser

Μεταβλητή	Default	Σημασία
net.slp.interfaces	τίποτα	Σε ποιες διευθύνσεις να ακούει το jSLP
net.slp.useScopes	default	Ποια scopes να χρησιμοποιούνται στα μηνύματα SLP
net.slp.DAAddresses	τίποτα	Λίστα με DAs που θα χρησιμοποιηθούν.
net.slp.noDADiscovery	ψευδές	Να μην γίνεται ανακάλυψη των DAs. Μόνο αν έχει επιλεχθεί το net.slp.DAAddresses.
net.slp.waitTime	1000	Καθυστέρηση μέχρι να ξεκινήσει η ανακάλυψη των DAs, κα.
net.slp.traceDATraffic	ψευδές	Καταγραφή της κίνησης προς τον DA.
net.slp.traceMsg	ψευδές	Καταγραφή των μηνυμάτων
net.slp.traceDrop	ψευδές	Καταγραφή των απορριφθέντων μηνυμάτων
net.slp.traceReg	ψευδες	Καταγραφή των καταχωρήσεων και των διαγραφών
net.slp.mulicastTTL	255	Το TTL που χρησιμοποιείται για multicast μηνύματα
net.slp.MTU	1400	Το μέγιστο μέγεθος ενός πακέτου UDP σε bytes.
net.slp.securityEnabled	ψευδές	Ενεργοποίησης της ασφάλειας.
net.slp.spi	τίποτα	Λίστα με τα SPIs που θα χρησιμοποιηθούν αν είναι ενεργοποιημένη η ασφάλεια.
net.slp.privateKey.SPI	τίποτα	Η τοποθεσία του ιδιωτικού κλειδιού για το SPI/
net.slp.publicKey.SPI	τίποτα	Η τοποθεσία του δημόσιου κλειδιού για το SPI.

Πίνακας 5.1: Πίνακας επιλογών για το jSLP

διαγράφει την υπηρεσία, εκτός κι αν ανανεωθεί η καταχώρηση. Το lifetime δίνεται ως ένας ακέραιος, που αντιπροσωπεύει δευτερόλεπτα. Κάποιες από τις προκαθορισμένες τιμές που μπορούν να δοθούν είναι ServiceURL.LIFETIME_DEFAULT και ServiceURL.LIFETIME_PERMANENT. Με την πρώτη τιμή επιλέγεται το lifetime που είναι διαλεγμένο στο jSLP, ενώ με τη δεύτερη η υπηρεσία δεν διαγράφεται ποτέ. Η δεύτερη τιμή έχει το πλεονέκτημα ότι καταργεί την ανάγκη περιοδικής ανανέωσης της καταχώρησης, αλλά έχει και τα μειονεκτήματα ότι αφενός όποιος πάρει μια αναφορά στην υπηρεσία μπορεί να θεωρήσει εσφαλμένα ότι η υπηρεσία θα είναι διαθέσιμη για πάντα κι αφετέρου αν η υπηρεσία σταματήσει απροσδόκητα μπορεί να μην ενημερωθεί ο Advertiser ότι χάθηκε και να συνεχίσει να παρέχει λανθασμένες πληροφορίες.

Η χρήση του Advertiser είναι πολύ απλή. Καλώντας τη μέθοδο register(ServiceURL url, Dictionary attributes) η υπηρεσία μας μπορεί να βρεθεί από όσους χρησιμοποιούν το SLP. Υπάρχει μια εμφανής αντιστοιχία μεταξύ των attributes του SLP και των properties του OSGi κι όντως επιλέξαμε να είναι τα ίδια. Υπάρχει η δυνατότητα να χρησιμοποιήσουμε και scopes, αλλά για την παρούσα υλοποίηση επιλέξαμε να την αγνοήσουμε. Σαν lifetime χρησιμοποιούμε το LIFETIME_DEFAULT, οπότε πρέπει περιοδικά να ανανεώνουμε την καταχώρηση.

Η χρήση του Locator είναι ελαφρώς πιο περίπλοκη. Η κύρια μέθοδος είναι η getServices() στην οποία περνάν σαν παράμετροι ο τύπος της υπηρεσίας, τα attributes και τα scopes και το πρόγραμμα αναλαμβάνει να βρει τις υπηρεσίες που ικανοποιούν όλα τα παραπάνω. Η μέθοδος επιστρέφει ένα αντικείμενο της κλάσης ServiceLocationEnumeration γιατί είναι δυνατόν να βρεθούν πολλές υπηρεσίες που να ικανοποιούν τα κριτήρια μας. Το αντικείμενο αυτό υλοποιεί το interface Enumeration της Java και μπορούμε να προσπελάσουμε τα στοιχεία του χρησιμοποιώντας τις μεθόδους hasMoreElements() και nextElement().

5.4 SOAP

Όπως έχουμε αναφέρει, για τις απομακρυσμένες κλήσεις χρησιμοποιούμε το SOAP. Το SOAP δεν απαιτεί κάποια ιδιαίτερη σχεδιαστική απόφαση (πέρα από το σχεδιασμό των web services), οπότε προχωράμε κατευθίαν στην υλοποίηση που διαλέξαμε.

5.4.1 kSOAP

Η πλήρης διανομή του Knopflerfish περιέχει δύο bundles που μπορούν α εξυπηρετούν web services: το kSOAP και το Axis. Και τα δυο bundles έχουν φτιαχτεί από την ομάδα του Knopflerfish κι ουσιαστικά αποτελούν wrappers για τα αντίστοιχα προγράμματα.

Το Axis είναι πιο δημοφιλές στο χώρο των web services, αλλά δυστυχώς το αντίστοιχο osgi bundle παρουσιάζει συχνά προβλήματα. Για αυτό το λόγο αναγκαστή-

καμε να χρησιμοποιήσουμε το kSOAP, το οποίο για το σκοπό που το χρειαζόμαστε δεν υπολείπεται σε τίποτα.

Το kSOAP 2 είναι μια υλοποίηση ανοικτού κώδικα του SOAP, στη γλώσσα Java. Μάλιστα, έχει σχεδιαστεί έτσι ώστε να έχει ελάχιστες απαιτήσεις και τρέχει σε J2ME. Δυστυχώς, επειδή δεν είναι πολύ δημοφιλές δεν υπάρχει αρκετό documentation.

Η δημιουργία web services στο Knopflerfish είναι πολύ απλή κι ακολουθεί το whiteboard pattern. Δημιουργούμε μια υπηρεσία όπως συνήθως, χωρίς να χρειάζεται κάποια αλλαγή επειδή είναι web service[1]. Μετά καταχωρούμε την υπηρεσία στο framework, δηλώνοντας το ειδικό property SOAP.service.name. Το kSOAP υλοποιεί έναν ServiceListener και κάθε φορά που βλέπει καταχώρηση υπηρεσίας με αυτό το property, κάνει αυτόματα την υπηρεσία διαθέσιμη ως web service. Αν υποθέσουμε ότι το property SOAP.service.name έχει την τιμή test τότε το web service θα είναι διαθέσιμο ως:

http://<address>:<port>/soap/services/test

Παρατηρούμε ότι η όλη διαδικασία είναι εξαιρετικά απλή: αντί να καλέσουμε κάποια υπηρεσία, απλά καταχωρούμε την δικιά μας υπηρεσία κι όλα τα υπόλοιπα γίνονται αυτόματα (αυτή είναι η ουσία του whiteboard pattern). Όμως πρέπει να προσέξουμε ότι δεν λαμβάνουμε κάποια επιβεβαίωση ότι όντως καταχωρήθηκε το αντίστοιχο web service.

5.5 Mediator

5.5.1 RemoteContext

Το RemoteContext παρέχει λειτουργίες παρόμοιες με το BundleContext, αλλά αναφέρεται σε δικτυακές υπηρεσίες. Όσες υπηρεσίες καταχωρούνται με το RemoteContext γίνονται διαθέσιμες σε απομακρυσμένα OSGi frameworks κι όσες υπηρεσίες ανακαλύπτονται είναι επίσης απομακρυσμένες.

Προσπαθήσαμε το RemoteContext να έχει κατά το δυνατόν παρόμοιο interface με το BundleContext. Στα Listings 5.1 και 5.2 φαίνονται οι σχετικές μέθοδοι.

```
BundleContext bc;

bc.registerService(String clazz, Object service, Properties
    props);
bc.registerService(String[] classes, Object service,
    Properties props);
bc.addServiceListener(ServiceListener listener);
bc.addServiceListener(ServiceListener listener, String
    filter);
```

[1]Υπάρχει ο περιορισμός ότι πρέπει να χρησιμοποιούνται μόνο αντικείμενα που είναι serializable

```
bc.removeServiceListener(ServiceListener listener);
```

Listing 5.1: Το interface του BundleContext

```
RemoteContext rc;

rc.registerService(String clazz, Object service, Properties
    props);
rc.registerService(String[] classes, Object service,
    Properties props);
rc.unregisterService(ServiceData sd);
rc.addServiceListener(RemoteServiceListener listener,
    String clazz);
rc.removeServiceListener(RemoteServiceListener listener,
    String clazz);
```

Listing 5.2: Το interface του RemoteContext

Μια διαφορά που δεν είναι ορατή παραπάνω είναι ότι η μέθοδος registerService στο BundleContext επιστρέφει ένα αντικείμενο του τύπου ServiceRegistration. Αν αργότερα ο χρήστης θελήσει να διαγράψει την υπηρεσία, καλεί ServiceRegistration.unregister(). Το αντικείμενο ServiceRegistration δεν είναι ιδιαιτέρα βολικό για τους σκοπούς μας. Δεν έχει κάποιον constructor, απαιτεί λειτουργίες που δεν μας είναι χρήσιμες και δεν είναι δυνατό να μαζέψουμε από αυτό όλες τις πληροφορίες σχετικές με μια απομακρυσμένη υπηρεσία. Για αυτό το λόγο δημιουργήσαμε το αντικείμενο ServiceData που έχει όλες τις πληροφορίες σχετικές με μια καταχώρηση. Η διαγραφή μιας υπηρεσίας γίνεται καλώντας rc.unregisterService().

Η άλλη διαφορά είναι ότι όταν δηλώνουμε έναν ServiceListener δηλώνουμε και ποια κλάση τον ενδιαφέρει, έτσι ώστε να μην καλείται για γεγονότα που αφορούν άλλες υπηρεσίες. Το BundleContext είναι πιο ευέλικτο κι επιτρέπει τη δήλωση οποιουδήποτε φίλτρου LDAP, που καλύπτει είτε την κλάση ή τα properties.

Στον mediator ένας listener καλείται όταν ανακαλυφθεί ή χαθεί μια υπηρεσία. Μια απομακρυσμένη υπηρεσία προσδιορίζεται από το ServiceURL της, οπότε αυτό πρέπει να δοθεί στον listener. Δυστυχώς αυτό καθιστά τον ServiceListener του OSGi ακατάλληλο κι αναγκαστήκαμε να δημιουργήσουμε τον δικό μας RemoteServiceListener. Οι διαφορές τους είναι ελάχιστες. Αντικαταστήσαμε το ServiceEvent με έναν ακέραιο, που δείχνει τον τύπο του γεγονότος. Οι τιμές είναι ίδιες με αυτές του ServiceEvent, δηλαδή το ServiceEvent.REGISTERED δείχνει ανακάλυψη υπηρεσίας και το ServiceEvent.UNREGISTERING δείχνει απώλεια υπηρεσίας (οι υπόλοιπες τιμές δεν χρησιμοποιούνται). Το ServiceEvent δεν μας βόλευε γιατί περιέχει αναφορά σε ServiceReference. Τα δύο interfaces φαίνονται παρακάτω:

```
public interface ServiceListener {
  public void serviceChanged(ServiceEvent event);
}
```

```
public interface RemoteServiceListener {
  public void serviceChanged(int event, ServiceURL url);
}
```

Listing 5.3: ServiceListener και RemoteServiceListener

Σημειώνουμε ότι το RemoteContext και το BundleContext είναι τελείως ανεξάρτητα. Μπορούμε να δηλώσουμε μια υπηρεσία μόνο στο RemoteContext ή μόνο στο BundleContext. Ηταν σχεδιαστική απόφαση να μην περιέχει το RemoteContext όλες τις υπηρεσίες που δηλώνονται στο framework γιατί μπορεί ο πάροχος μιας υπηρεσίας να κρίνει, για οποιοδήποτε λόγο, ότι δε θέλει η υπηρεσία του να είναι διαθέσιμη σε άλλα frameworks.

Όταν μια υπηρεσία καταχωρείται στο RemoteContext ελέγχεται ότι το αντικείμενο όντως υλοποιεί τις κλάσεις που ισχυρίζεται ότι υλοποιεί. Αυτός ο έλεγχος γίνεται κι από το framework όταν δηλώνεται σε αυτό μια υπηρεσία και μάλιστα, χρησιμοποιήθηκαν μέρη του κώδικα του Knopflerfish. Ίσως φαίνεται περιττό να γίνεται ο ίδιος έλεγχος δυο φορές, αλλά δεν υπάρχει κάποιος τρόπος να ξέρει το RemoteContext ότι όντως έγινε ο έλεγχος.

5.5.2 RemoteContext και SLP

Όταν δηλώνουμε μια υπηρεσία στο RemoteContext τότε αυτή γίνεται ορατή σε αιτήσεις SLP. Περιγράψαμε προηγουμένως τι μορφή έχει το SLP ServiceType για την κάθε υπηρεσία. Επίσης, όταν δηλώνεται κάποιος ServiceListener τότε στέλνονται αιτήσεις SLP για να ανακαλυφθεί η αντίστοιχη υπηρεσία.

Το καθοριστικό σημείο του SLP είναι ότι δεν υποστηρίζει announcements. Δηλαδή, όταν μια νέα υπηρεσία εμφανίζεται στο δίκτυο ή όταν μια υπηρεσία εξαφανίζεται, δεν στέλνεται κάποιο multicast μήνυμα για να ειδοποιηθούν όλοι οι υπόλοιποι για το γεγονός. Αυτό μας αναγκάζει να πραγματοποιούμε περιοδικά SLP queries για να έχουμε ενημερωμένες πληροφορίες για τις υπηρεσίες στο δίκτυο.

Στην υλοποίησή μας υπάρχει ένα thread το οποίο στέλνει μηνύματα για ανακάλυψη κάθε 20 δευτερόλεπτα (καθορίζεται από τη μεταβλητή DISCOVERY_INTERVAL). Διατηρούμε ένα ευρετήριο με τις γνωστές υπηρεσίες κι αν ανακαλυφθεί κάποια καινούργια, τότε προστίθεται σε ευρετήριο κι ελέγχεται αν πρέπει να ειδοποιηθεί κάποιος ServiceListener. Επίσης, αν κάποια από τις γνωστές υπηρεσίες δεν εμφανιστεί στις απαντήσεις, τότε θεωρείται ότι εξαφανίστηκε, διαγράφεται και ειδοποιούνται οι τυχόν ServiceListeners. Με αυτό το σύστημα μπορεί να περάσουν έως και 20 δευτερόλεπτα μέχρι να ειδοποιηθεί ένας listener για κάποιο γεγονός. Ως μια απλή βελτίωση, όταν δηλώνεται ένας listener ελέγχεται αν η υπηρεσία που ζητάει είναι ήδη γνωστή κι ειδοποιείται κατευθείαν.

Όταν γίνεται αναζήτηση για υπηρεσίες τότε χρησιμοποιούμε το service type service:osgi αντί να ψάχνουμε χωριστά για κάθε υπηρεσία (πχ service:osgi:test_Service). Δεδομένου ότι πρέπει ουσιαστικά να κάνουμε polling, θα επιβαρύναμε πολύ το δί-

κτυο αν το κάναμε χωριστά για κάθε υπηρεσία. Το μόνο πιθανό μειονέκτημα είναι ότι ίσως λάβουμε πολύ περισσότερες απαντήσεις από όσες θα λαμβάναμε για μια πιο συγκεκριμένη αναζήτηση, αλλά όλες οι απαντήσεις αποθηκεύονται και μπορεί να χρησιμεύσουν μελλοντικά.

Τέλος, σημειώνουμε ότι ο τρόπος που χρησιμοποιούμε τα service urls είναι εν μέρει παράδοξος, γιατί ένα service url που θα λάβουμε ως απάντηση δεν δείχνει κάποια υπηρεσία που θα καλέσουμε. Από το service url εξάγουμε το όνομα της υπηρεσίας και τη διεύθυνση και τελικά φτιάχνουμε ένα νέο URL για τη SOAP κλήση.

5.5.3 Endpoints

Οι κλήσεις των υπηρεσιών γίνονται πάνω από SOAP. Όπως ήδη αναφέραμε σαν εξυπηρετητή χρησιμοποιούμε το kSOAP.

Σε κάθε OSGi framework καταχωρούμε ένα web service που εξυπηρετεί όλες τις κλήσεις. Επίσης, σε κάθε framework δηλώνουμε και μια υπηρεσία που δημιουργεί τις SOAP κλήσεις. Επιλέξαμε να χρησιμοποιούμε μόνο ένα web service επειδή αντιμετωπίσαμε δυσκολίες στην αυτόματη δημιουργία proxy κλάσεων που θα μετέτρεπαν τις απλές κλήσεις σε SOAP κλήσεις.

Ο χρήστης για να κάνει μια απομακρυσμένη κλήση παίρνει μια αναφορά στην υπηρεσία LocalEndpoint και μετά καλεί την μέθοδο invokeMethod. Υποτίθεται ότι έχει προηγηθεί η ανακάλυψη της υπηρεσίας και ο χρήστης ξέρει το ServiceURL της. Το LocalEndpoint χρησιμοποιεί το kSOAP για να καλέσει την αντίστοιχη μέθοδο invokeMethod του RemoteEndpoint. Το RemoteEndpoint ελέγχει το όνομα της επιθυμητής υπηρεσίας κι ελέγχει αν έχει καταχωρηθεί με το RemoteContext. Σε αυτή την περίπτωση καλεί την ζητούμενη μέθοδο με τις παραμέτρους που έχουν δοθεί. Αυτό γίνεται χρησιμοποιώντας τις δυνατότητες του reflection που διαθέτει η γλώσσα Java. Στη συνέχεια, το RemoteEndpoint στέλνει το αποτέλεσμα στο LocalEndpoint κι αυτό με τη σειρά του στον χρήστη. Σημειώνουμε ότι το RemoteEndpoint δεν χρειάζεται κάποια ειδική σχεδίαση για να δηλωθεί ως web service.

Επειδή υπάρχει μόνο μια κλάση για όλες τις υπηρεσίες, αναγκαστικά αυτή είναι πολύ γενική στη μορφή της. Παρακάτω φαίνονται οι κλάσεις LocalEndpoint και RemoteEndpoint:

```
public class LocalEndpoint {
  public Object invokeMethod(ServiceURL url, String
      methodName, Vector params);
}

public class RemoteEndpoint {
  public Object invokeMethod(String service, String
      methodName, Vector params);
}
```

Listing 5.4: Τα δύο endpoints του SOAP

Ο χρήστης για να κάνει μια απομακρυσμένη κλήση παίρνει μια αναφορά στην υπηρεσία LocalEndpoint και μετά καλεί την μέθοδο invokeMethod. Υποτίθεται ότι έχει προηγηθεί η ανακάλυψη της υπηρεσίας και ο χρήστης ξέρει το ServiceURL της. Το LocalEndpoint χρησιμοποιεί το kSOAP για να καλέσει την αντίστοιχη μέθοδο invokeMethod του RemoteEndpoint. Το RemoteEndpoint ελέγχει το όνομα της επιθυμητής υπηρεσίας κι ελέγχει αν έχει καταχωρηθεί με το RemoteContext. Σε αυτή την περίπτωση καλεί την ζητούμενη μέθοδο με τις παραμέτρους που έχουν δοθεί. Αυτό γίνεται χρησιμοποιώντας τις δυνατότητες του reflection που διαθέτει η γλώσσα Java. Στη συνέχεια, το RemoteEndpoint στέλνει το αποτέλεσμα στο LocalEndpoint κι αυτό με τη σειρά του στον χρήστη. Σημειώνουμε ότι ενώ το LocalEndpoint πρέπει να ασχοληθεί με τη δημιουργία της SOAP κλήσης, το RemoteEndpoint αναπτύσσεται ως συνηθισμένη κλάση Java και όταν δηλώνεται ως web service το kSOAP αναλαμβάνει όλα τα υπόλοιπα.

Φυσικά, αφού μιλάμε για SOAP κλήσεις πρέπει να θυμόμαστε ότι όλοι οι παράμετροι των κλήσεων πρέπει να ανήκουν σε serializable κλάσεις. Σε αυτές ανήκουν (μεταξύ άλλων) οι απλές κλάσεις Integer, Float, String, αλλά και τα primitive types int, boolean, κλπ. Οι υπόλοιπες κλάσεις χρειάζονται έναν serializer για να δουλέψουν με το SOAP.

Κεφάλαιο 6

Ιδέες - Προτάσεις

Ο mediator αναπτύχθηκε στα πλαίσια διπλωματικής εργασίας στο Πολυτεχνείο και τα χρονικά περιθώρια περιόρισαν τις λειτουργίες που υλοποιήσαμε. Υπάρχουν πολλές ιδέες για μελλοντική ανάπτυξη. Παρακάτω παραθέτουμε τις σημαντικότερες από αυτές.

Χρήση SLP scopes

Τα scopes είναι ένα χαρακτηριστικό του SLP που επιτρέπουν την κλιμάκωσή του σε μεγάλες εγκαταστάσεις με πολλούς κόμβους. Με τα scopes μπορούμε ουσιαστικά να διαχωρίσουμε το δίκτυο σε μικρότερα υποδίκτυα και να περιορίσουμε τα μηνύματα που ανταλλάσσονται μεταξύ τους. Το jSLP υποστηρίζει πλήρως τα scopes και λογικά και ο mediator με μικρές αλλαγές θα μπορούσε να τα υποστηρίξει.

Αναζήτηση με βάση τα attributes

Στην τρέχουσα υλοποίηση του mediator τα attributes των υπηρεσιών που καταχωρούνται στο RemoteContext γίνονται διαθέσιμα μέσω SLP. Θα μπορούσε το κομμάτι της αναζήτησης να αλλαχθεί έτσι ώστε να λαμβάνει υπόψιν και τα attributes. Όπως όταν κάποιος καταχωρεί έναν ServiceListener στο BundleContext μπορεί να δώσει ένα LDAP φίλτρο για τα properties που τον ενδιαφέρει, θα μπορούσε να δίνει το αντίστοιχο φίλτρο στο RemoteContext για να περιορίσει το εύρος των απαντήσεων που ζητάει.

Υλοποίηση BundleListener

Με την παρούσα υλοποίηση του mediator για να αφαιρεθεί μια υπηρεσία ή ένας listener από τη μνήμη του RemoteContext πρέπει να κληθεί η αντίστοιχη συνάρτηση. Αν όμως ένας χρήστης παραλείψει να καλέσει αυτές τις συναρτήσεις τότε στη μνήμη του mediator θα παραμείνουν αναφορές σε αντικείμενα που δεν είναι πλέον ενεργά. Αυτό έχει δυσάρεστες συνέπειες τόσο στη συνέπεια των απαντήσεων που δίνει ο mediator όσο και στη ποσότητα της μνήμης που χρησιμοποιεί.

Η λύση σε αυτό το πρόβλημα είναι αυτή που χρησιμοποιεί και το ίδιο το framework. Όταν καταχωρείται ένα αντικείμενο στον mediator σημειώνουμε από ποιο bundle προέρχεται. Υλοποιώντας έναν BundleListener ενημερωνόμαστε για τις αλλαγές στα bundles κι αν ένα bundle σταματηθεί τότε αφαιρούμε τα αντικείμενα που προέρχονται από αυτό.

Δυναμική δημιουργία proxy κλάσεων

Η υλοποίηση μας έχει μόνο ένα web service σε κάθε άκρο, που αναλαμβάνει να εξυπηρετήσει όλες τις υπηρεσίες. Μια άλλη προσέγγιση θα ήταν να καταχωρούμε όλες τις υπηρεσίες του RemoteContext ως web services. Αυτό θα απαιτούσε τη δυναμική δημιουργία μιας κλάσης για την κάθε υπηρεσία, που θα έχει το ίδιο interface με την υπηρεσία και θα μετατρέπει την κάθε κλήση σε αυτή σε κλήση SOAP.

Αυτός ήταν ένας από τους αρχικούς στόχους της εργασίας αλλά εγκαταλείφθηκε λόγω δυσκολιών. Ένας πιθανός τρόπος υλοποίησης θα ήταν με χρήση του εργαλείου WSDL2Java του Axis. Το Axis παρέχει αυτόματα το WSDL για κάθε web service, οπότε έχουμε έτοιμη την είσοδο στο WSDL2Java. Ωστόσο, λαμβάνοντας υπόψιν ότι το Axis bundle παρουσίαζε προβλήματα, το WSDL2Java δεν ήταν άμεσα διαθέσιμο από το bundle και η έξοδος του WSDL2Java είναι σε μορφή κειμένου και θα έπρεπε να μεταφραστεί, τελικά επιλέξαμε να υλοποιήσουμε την εναλλακτική προσέγγιση.

Καλύτερη αντιμετώπιση των σφαλμάτων στις SOAP κλήσεις

Αν το RemoteEndpoint αντιμετωπίσει οποιοδήποτε πρόβλημα τότε επιστρέφει απλώς null. Φυσικά, υπάρχουν πολλά διαφορετικά είδη σφαλμάτων και ίσως θα ήταν χρήσιμο για τον πελάτη να γνωρίζει περισσότερες λεπτομέριες για το σφάλμα. Για παράδειγμα, το σφάλμα της κλήσης μιας υπηρεσίας που δεν υπάρχει είναι τελείως διαφορετικό από το σφάλμα της λανθασμένης κλήσης μιας υπηρεσίας. Το SOAP επιτρέπει την επιστροφή σφαλμάτων κι αυτή η δυνατότητα θα μπορούσε να χρησιμοποιηθεί για να επιστραφεί μια περιγραφή του σφάλματος.

Δυνατότητα ορισμού Serializer

Το SOAP μπορεί να δεχτεί αυτούσιους μόνο ορισμένους τύπους δεδομένων και για τους υπόλοιπους απαιτείται ο ορισμός κάποιου serializer. Η υλοποίηση μας σιωπηλά δέχεται μόνο τους προκαθορισμένους τύπους δεδομένων, ενώ αν προσπαθήσουμε να χρησιμοποιήσουμε κάποιον άλλο τύπο θα προκαλέσουμε μερικά Exceptions. Αυτό περιορίζει σημαντικά τις υπηρεσίες που μπορούν να αναπτυχθούν. Εφόσον ο προγραμματιστής μιας υπηρεσίας γράψει κάποιο Serializer για τους τύπους που τον ενδιαφέρουν θα έπρεπε να έχει τη δυνατότητα να το χρησιμοποιήσει.

Δημιουργία HTTP interface

Το OSGi framework καθιστά πολύ εύκολη την ανάπτυξη servlets. Θα έπρεπε να χρησιμοποιήσουμε αυτή τη δυνατότητα για να δημιουργησουμε ένα εύχρηστο interface για το mediator. Ο διαχειριστής θα μπορούσε να έχει πρόσβαση σε πληροφορίες σχετικές με τον mediator, όπως υπηρεσίες και listeners που έχουν καταχωρηθεί και υπηρεσίες που έχουν ανακαλυφθεί. Επίσης, το interface θα μπορούσε να παρουσιάζει τα logs τόσο του mediator, όσο και των jSLP και kSOAP. Τέλος, θα μπορούσε να υπάρχει η δυνατότητα να επέμβει ο διαχειριστής στα εσωτερικά του mediator, παραδείγματος χάριν σβήνοντας κάποιες καταχωρήσεις που θεωρεί ότι είναι πλέον άκυρες.

Κεφάλαιο 7

Επίλογος

Σε αυτή την εργασία σχεδιάσαμε και υλοποιήσαμε ένα σύστημα ανακάλυψης και κλήσης υπηρεσιών σε περιβάλλον τοπικού δικτύου. Το σύστημα λειτουργεί πάνω στην πλατφόρμα OSGi, το οποίο προσφέρει τη δυνατότητα εγκατάστασης, εκκίνησης και σταματήματος των υπηρεσιών χωρίς την ανάγκη επανεκκίνησης όλης της πλατφόρμας.

Οι σχεδιαστικές αποφάσεις της χρήσης των SLP και SOAP ήταν σχετικά εύκολες· ειδικά το SOAP φαίνεται να έχει επικρατήσει στον τομέα του. Το SLP έχει μεγαλύτερο ανταγωνισμό, αλλά η απλότητά του ήταν ο καθοριστικός παράγοντας, αφού οι υπόλοιπες λύσεις προσπαθούν να καλύψουν πολλά περισσότερα από την ανακάλυψη υπηρεσιών. Σε κάθε περίπτωση, η μελέτη των εναλλακτικών λύσεων ήταν συναρπαστική και οδήγησε σε βαθύτερη κατανόηση των θεμάτων.

Η υλοποίηση ήταν το πιο δύσκολο μέρος και κυρίως το κομμάτι που αφορούσε το OSGi. Δεν υπάρχει κάποια εγγενής δυσκολία χρήσης στην πλατφόρμα, αλλά το πρόβλημα ήταν η σχεδόν πλήρης έλλειψη τεκμηρίωσης. Αν και υπάρχουν αρκετές ελεύθερες υλοποιήσεις του OSGi, αυτές συνοδεύονται από ελάχιστα έγγραφα και στην περίπτωση του Knopflerfish παραπέμπουν σε εμπορικές λύσεις για υποστήριξη. Αυτό σήμαινε ότι αρκετές φορές έπρεπε να βγάλουμε μόνοι μας άκρη, κοιτώντας το πρότυπο του OSGi και τον πηγαίο κώδικα. Ωστόσο, πιστεύουμε ότι το OSGi είναι αρκετά αξιόλογο κι ελπίζουμε να αποκτήσει την αναγνώριση που του αξίζει.

Συνολικά, η εργασία ήταν πολύ ενδιαφέρουσα και προσέφερε πολύτιμες γνώσεις τόσο στις δικτυακές υπηρεσίες, όσο και στην ανάπτυξη εφαρμογών στο OSGi.