NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Distributed filtering and dissemination of XML data in peer-to-peer systems

Spyridoula K. Miliaraki

ATHENS

JULY 2011

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Κατανεμημένη Διήθηση και Διάχυση XML Δεδομένων σε Συστήματα Ομότιμων Κόμβων

Σπυριδούλα Κ. Μηλιαράκη

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2011

PhD THESIS

Distributed filtering and dissemination of XML data in peer-to-peer systems

Spyridoula K. Miliaraki

SUPERVISOR: Manolis Koubarakis, Professor UoA

THREE-MEMBER ADVISORY COMMITTEE:
Manolis Koubarakis, Professor UoA
Yannis Ioannidis, Professor UoA
Alex Delis, Professor UoA

SEVEN-MEMBER EXAMINATION COMMITTEE

Manolis Koubarakis,
Professor UoA

Yannis Ioannidis,
Professor UoA

Alex Delis,
Professor UoA

Mema Roussopoulos,
Assistant Professor UoA

Christos Tryfonopoulos,
Lecturer, University of Peloponnese

Peter Triantafillou,
Professor, University of Patras

Minos Garofalakis,
Professor, Technical University of Crete

Examination Date 21/07/2011

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Κατανεμημένη Διήθηση και Διάχυση XML Δεδομένων
σε Συστήματα Ομότιμων Κόμβων

Σπυριδούλα Μηλιαράκη

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Εμμανουήλ Κουμπαράκης, Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:
Εμμανουήλ Κουμπαράκης, Καθηγητής ΕΚΠΑ
Ιωάννης Ιωαννίδης, Καθηγητής ΕΚΠΑ
Αλέξιος Δελής, Καθηγητής ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Εμμανουήλ Κουμπαράκης,
Καθηγητής ΕΚΠΑ

Ιωάννης Ιωαννίδης,
Καθηγητής ΕΚΠΑ

Αλέξιος Δελής,
Καθηγητής ΕΚΠΑ

Δήμητρα-Ισιδώρα Ρουσσοπούλου,
Επίκουρη Καθηγήτρια ΕΚΠΑ

Χρήστος Τρυφωνόπουλος,
Λέκτορας Πανεπιστημίου Πελοποννήσου

Παναγιώτης Τριανταφύλλου,
Καθηγητής Πανεπιστημίου Πάτρας

Μίνως Γαροφαλάκης,
Καθηγητής Πολυτεχνείου Κρήτης

Ημερομηνία εξέτασης 21/07/2011

ABSTRACT

Publish/subscribe systems have emerged in recent years as a promising paradigm for offering various popular notification services such as news monitoring, e-commerce site monitoring and alerting services for digital libraries. Since XML is widely used as the standard format for data exchange on the Web, a lot of research has focused on designing efficient and scalable XML filtering systems. To offer XML filtering functionality on Internet-scale and avoid the typical problems of centralized solutions, we need to deploy such a service in a distributed environment.

In this thesis, we design, develop and evaluate an XML filtering system called FoXtrot. Our proposal combines the strengths of automata for fast XML filtering and distributed hash tables for building a fully distributed scalable system. In FoXtrot, we distribute an nondeterministic finite automaton on top of Pastry DHT exploiting the inherent parallelism of an NFA that allows it to be in several states at the same time. Structural matching is performed using the automaton, while we study different methods for also distributing the task of predicate evaluation. As a result, FoXtrot scales both for a large number of queries and a large number of predicates per query.

We extensively evaluate our system under various conditions and demonstrate that it can index millions of user queries exhibiting a high indexing and filtering throughput. At the same time FoXtrot achieves very good load balancing properties, improves its performance as we increase the size of the network and exhibits a sufficient degree of fault-tolerance. Our evaluation was done in a controlled environment of a local cluster and on the worldwide testbed provided by the PlanetLab network representing the real-world conditions of the Internet.

SUBJECT AREA: Distributed data filtering

KEYWORDS: peer-to-peer network, information filtering, distributed processing, XML data, non-deterministic finite automaton

ΠΕΡΙΛΗΨΗ

Τα συστήματα δημοσιεύσεων/συνδρομών (publish/subscribe systems) ή αλλιώς τα συστήματα διήθησης δεδομένων (filtering systems) αποτελούν μια ευρέως διαδεδομένη πρόταση στις μέρες μας. Τέτοια συστήματα έχουν διάφορες ενδιαφέρουσες και χρήσιμες εφαρμογές όπως οι ειδοποιήσεις για ειδήσεις ή δημοσιεύσεις ψηφιακών βιβλιοθηκών αλλά και για οποιαδήποτε άλλη πληροφορία μπορεί να ενδιαφέρεται ένας χρήστης. Δεδομένου ότι το μοντέλο XML έχει καθιερωθεί για την μορφοποίηση και την ανταλλαγή δεδομένων στο διαδίκτυο, μεγάλο μέρος της έρευνας στην περιοχή αυτή έχει επικεντρωθεί στον σχεδιασμό αποδοτικών συστημάτων για την διήθηση δεδομένων XML. Για να επιτύχουμε αποδοτική διήθηση δεδομένων XML σε μεγάλη κλίμακα και να αποφύγουμε τα συνήθη προβλήματα που εμφανίζουν οι συγκεντρωτικές προσεγγίσεις, πρέπει να οδηγηθούμε στην ανάπτυξη μιας τέτοια υπηρεσίας σε ένα κατανεμημένο περιβάλλον.

Στα πλαίσια της διατριβής, σχεδιάσαμε, υλοποιήσαμε και αξιολογήσαμε πειραματικά ένα κατανεμημένο σύστημα που ονομάστηκε FoXtrot. Η πρόταση μας συνδυάζει ένα μη αιτιοκρατικό αυτόματο για αποδοτική διήθηση δεδομένων XML καθώς και τους κατανεμημένους πίνακες κατακερματισμού (distributed hash tables) για ένα πλήρως κατανεμημένο και κλιμακωτό σύστημα. Η βασική ιδέα αυτής της εργασίας είναι η κατανομή του αυτόματου στους κόμβους του δικτύου και η εκτέλεση παράλληλων ανεξάρτητων υπολογισμών που μας προσφέρει η ιδιότητα της μη-αιτιοκρατίας. Εκτός από τις τεχνικές που παρουσιάζονται για το δομικό ταίριασμα (structural matching) των δεδομένων XML και για τις οποίες χρησιμοποιούμε το αυτόματο που περιγράψαμε, προτείνουμε και διάφορες εναλλακτικές τεχνικές για το ταίριασμα των τιμών (value matching) που περιλαμβάνονται στα δεδομένα XML με στόχο την κατανομή του φόρτου και αυτής της εργασίας.

Τέλος, πραγματοποιήσαμε μια εκτενή πειραματική αξιολόγηση του συστήματος FoXtrot τόσο σε ένα ελεγχόμενο περιβάλλον όσο και σε ένα ρεαλιστικό περιβάλλον μεγάλης κλίμακας που παρέχεται από το PlanetLab. Με βάση την αξιολόγηση προκύπτει ότι το σύστημα FoXtrot επιτυγχάνει υψηλές επιδόσεις τόσο για την αποθήκευση των ερωτήσεων στο κατανεμημένο αυτόματο όσο και για την διήθηση και διάχυση των δεδομένων XML. Επιπλέον, είναι χαρακτηριστικό της δυνατότητας κλιμάκωσης του συστήματος ότι η αύξηση του μεγέθους του δικτύου που εκτελείται το FoXtrot οδηγεί σε αισθητή βελτίωση της απόδοσης του συστήματος.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κατανεμημένη διήθηση δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: δίκτυα ομότιμων κόμβων, διήθηση δεδομένων, κατανεμημένη επεξεργασία, δενδρικά δεδομένα, μη αιτιοκρατικό αυτόματο

To Mihalis, for giving me back my faith.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor, Manolis Koubarakis for trusting me and supporting me throughout the years of my PhD. I would also like to thank the members of my doctoral committee, Yannis Ioannidis, Alex Delis, Mema Roussopoulos, Peter Triantafillou, Minos Garofalakis and Christos Tryfonopoulos for their comments and suggestions for improving and extending this work.

Special thanks go to the rest of my colleagues in the group, Zoi, Matoula, Kostis and Babis for being good workmates. Especially Zoi, apart from a close friend, was always there for helping me and advising me at every step of this journey.

Special thanks go to my parents, Kostas and Sofia, my brother Stelios and my sister Alkmini for their love, patience and encouragement throughout these years. My father's belief in me always motivated me for never giving up despite the difficulties.

This thesis became possible as special persons stood by me in difficult personal moments and offered a relief after long working hours and work-spoiled weekends. Words are not enough to thank Mihalis for being part of my life, for his love and support to all my decisions.

Throughout this thesis I am very grateful and honored to have received financial support by Microsoft Research through its European PhD Scholarship Programme. I would like to thank all the people involved in this research grant.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

1. Εισαγωγή

Η ραγδαία ανάπτυξη του διαδικτύου έχει οδηγήσει στην ευρεία διαθεσιμότητα ενός μεγάλου όγκου δεδομένων για τους χρήστες. Μέσα σε αυτή την πληθώρα δεδομένων έχει γίνει δυσκολότερο για τους χρήστες να αναζητήσουν από μόνοι τους και να ανακαλύψουν τις πληροφορίες που τους ενδιαφέρουν. Για να αντιμετωπιστεί αυτή η πρόκληση, έχουνε προταθεί και διαδοθεί τα τελευταία χρόνια τα ονομαζόμενα συστήματα δημοσιεύσεων/συνδρομών (publish/subscribe systems) ή αλλιώς συστήματα διήθησης πληροφορίας (information filtering systems) ως μια λύση στο παραπάνω πρόβλημα. Ένα σύστημα δημοσιεύσεων/συνδρομών δίνει την δυνατότητα στους χρήστες να εκφράζουν τα ενδιαφέροντα τους υποβάλλοντας τα με την μορφή κάποιου ερωτήματος (continuous query) ή αλλιώς υποβάλλοντας μια συνδρομή (subscription). Στην συνέχεια μόλις γίνουν διαθέσιμα δεδομένα που ταιριάζουν στα ενδιαφέροντα κάποιων χρηστών, το σύστημα αναλαμβάνει να τους ενημερώσει. Με αυτόν τον τρόπο ο χρήστης δεν χρειάζεται να αναζητεί από μόνος του την πληροφορία που τον ενδιαφέρει αλλά να την εκφράζει μέσω κάποιων ερωτημάτων ή κάποιου προφίλ και το σύστημα είναι αυτό που θα τον ενημερώσει μόλις βρεθεί κάτι που ταιριάζει στα ενδιαφέροντα του.

1.1 Συστήματα δημοσιεύσεων/συνδρομών

Τα συστήματα δημοσιεύσεων/συνδρομών έχουνε χρησιμοποιηθεί σε πλήθος εφαρμογών όπως είναι για παράδειγμα η παρακολούθηση ειδήσεων ή ιστολογίων καθώς και η ενημέρωση από ψηφιακές βιβλιοθήκες. Για παράδειγμα, θεωρήστε έναν χρήστη που θέλει να ενημερωθεί για τις νέες δημοσιεύσεις ενός συγκεκριμένου συγγραφέα ή για τις νέες δημοσιεύσεις σε κάποιο συγκεκριμένο γνωστικό πεδίο. Ένας χρήστης μπορεί να εκφράσει τα ενδιαφέροντα του χρησιμοποιώντας διαφορετικής πολυπλοκότητας μοντέλα όπως μπορεί να είναι ένα σύνολο από λέξεις κλειδιά (keywords) ή ένα σύνολο επερωτήσεων σε κάποια γλώσσα επερωτήσεων όπως η XPath. Στα πλαίσια αυτής της διατριβής έχουμε επιλέξει το δενδρικό μοντέλο δεδομένων της γλώσσας XML (eXtensible Markup Language).

1.2 Μοντέλο δεδομένων XML

Το μοντέλο XML είναι ένα ευρέως διαδεδομένο μοντέλο για την ανταλλαγή δεδομένων στο διαδίκτυο και για αυτό μεγάλο μέρος των εργασιών που αφορούν συστήματα δημοσιεύσεων/συνδρομών έχουν επικεντρωθεί στο συγκεκριμένο μοντέλο. Σε αυτά τα συστήματα λοιπόν, οι χρήστες υποβάλλουν τα ερωτήματα τους εκφρασμένα σε μία γλώσσα επερωτήσεων όπως είναι η γλώσσα XPath (XML Path language) ή η γλώσσα XQuery (XML Query language) και τα δεδομένα περιγράφονται χρησιμοποιώντας το μοντέλο της XML. Οπότε ένα σύστημα φιλτραρίσματος δεδομένων XML πρέπει να *ταιριάξει* τα ερωτήματα των χρηστών με τα δεδομένα που δημοσιεύονται σε μορφή XML εγγράφων. Η σχετική βιβλιογραφία σε αυτήν την περιοχή έχει επικεντρωθεί στην αποδοτική εκτέλεση της διαδικασίας ταιριάσματος.

2. Συστήματα δημοσιεύσεων/συνδρομών XML δεδομένων

Τα τελευταία χρόνια, έχουν προταθεί στην σχετική βιβλιογραφία διάφορα κεντρικά συστήματα όπως είναι το σύστημα YFilter και το σύστημα XTrie που στοχεύουν στο αποτελεσματικό φιλτράρισμα των XML δεδομένων έναντι ενός μεγάλου συνόλου από ερωτήματα χρηστών. Ωστόσο, προκειμένου να προσφέρει κανείς ικανοποιητική απόδοση σε μεγάλη κλίμακα θα πρέπει να αποφύγει τα συνήθη μειονεκτήματα των κεντρικοποιημένων προσεγγίσεων όπως είναι η ύπαρξη ενός μοναδικού σημείου αποτυχίας (single point of failure), η δυσκολία κλιμάκωσης (scalability), καθώς

αυξάνεται το πλήθος ερωτημάτων των χρηστών και το πλήθος των XML εγγράφων που δημοσιεύονται, καθώς και η πιθανή πρόκληση συμφόρησης (bottleneck). Βάσει λοιπόν αυτών των μειονεκτημάτων που αναφέραμε προκειμένου να εξασφαλίσουμε ένα κλιμακώσιμο σύστημα δημοσιεύσεων/συνδρομών οδηγούμαστε στο να σχεδιάσουμε ένα πλήρως κατανεμημένο σύστημα.

Οι προτάσεις για κατανεμημένα συστήματα δημοσιεύσεων/συνδρομών μειοψηφούν σε σχέση με τις αντίστοιχες κεντρικοποιημένες αλλά ενδεικτικά παραδείγματα συστημάτων αποτελούν το σύστημα XNet και το σύστημα ONYX. Τα προηγούμενα συστήματα «χτίζονται» πάνω σε ένα δίκτυο αποτελούμενο από κόμβους που αποτελούν και τους δρομολογητές των XML δεδομένων που φτάνουν στο δίκτυο. Οι δρομολογητές είναι υπεύθυνοι για την διαβίβαση των XML δεδομένων προς τους υπόλοιπους κόμβους με στόχο να φτάσουν τελικά στους ενδιαφερόμενους χρήστες/συνδρομητές. Ας πάρουμε για παράδειγμα το σύστημα ONYX όπου οι κόμβοι που συμμετέχουν οργανώνονται σε μια δενδρική δομή, την οποία χρησιμοποιούν οι κόμβοι για να επικοινωνήσουν μεταξύ τους και να ανταλλάξουν δεδομένα. Για το φιλτράρισμα των δεδομένων ο κάθε κόμβος διατηρεί τοπικά ένα κεντρικοποιημένο σύστημα δημοσιεύσεων/συνδρομών (συγκεκριμένα στην περίπτωση του συστήματος ONYX χρησιμοποιείται ένα αυτόματο που αντιστοιχεί στην δομή του κεντρικού συστήματος YFilter). Αυτό το αυτόματο χρησιμοποιείται για να αναπαραστήσει τα ερωτήματα που αποθηκεύονται σε κάθε κόμβο και κατά την διάρκεια της δρομολόγησης ο κάθε κόμβος εκτελεί το τοπικό του αυτόματο ώστε να καθορίσει σε ποιους κόμβους πρέπει να προωθηθούν τα δεδομένα.

Μια σημαντική απόφαση σε κάθε κατανεμήμενο σύστημα δημοσιεύσεων/συνδρομών όπως τα παραπάνω είναι ο τρόπος κατανομής των ερωτημάτων ανάμεσα στους κόμβους του δικτύου. Ο τρόπος που θα γίνει αυτή η κατανομή των ερωτημάτων θα επηρεάσει τις διαδρομές που θα ακολουθούν τα XML δεδομένα που δημοσιεύονται στο δίκτυο και ακολούθως θα επηρεάζουν συνολικά την απόδοση του συστήματος. Αν για παράδειγμα ο φόρτος μεταξύ των κόμβων είναι σε μεγάλο βαθμό άνισος (για παράδειγμα αν υπάρχει κάποιος κόμβος που τα ερωτήματα που αποθηκεύει τοπικά «ικανοποιούνται» πολύ συχνότερα από ότι τα ερωτήματα των άλλων κόμβων), η απόδοση του συστήματος θα επιβαρυνθεί. Συνολικά, ο φόρτος του κάθε κόμβου περιλαμβάνει διάφορες εργασίες όπως είναι η αποθήκευση και ευρετηρίαση των ερωτημάτων, η διήθηση ή αλλιώς φιλτράρισμα των εισερχόμενων XML δεδομένων, καθώς και η ενημέρωση των ενδιαφερόμενων χρηστών όταν κάποιο ερώτημα τους ικανοποιηθεί.

Στην περίπτωση που οι κόμβοι είναι οργανωμένοι σε μία δενδρική δομή, όπως στην περίπτωση του συστήματος ONYX, οι κόμβοι που θα βρίσκονται πιο κοντά στην «ρίζα» αυτής της ιεραρχικής δομής θα υποφέρουν μεγαλύτερο φόρτο ενώ οι υπόλοιποι μικρότερο. Αυτή η άνιση κατανομή του φόρτου καθώς αυξάνει το πλήθος των εισερχόμενων δεδομένων καθώς και η ταχύτητα δημοσίευσης τους μπορεί να οδηγήσει σε υπερφόρτωση των κόμβων κοντά στην «ρίζα» και επομένως και του συστήματος. Επιπλέον, στην περίπτωση του συστήματος ONYX, οι συγγραφείς χρησιμοποιούν ένα κεντρικό σημείο ελέγχου που αποφασίζει που θα αποθηκευτεί το κάθε ερώτημα. Επομένως για κάθε νέο ερώτημα, πρέπει να υπάρξει επικοινωνία με αυτόν τον κεντρικό εξυπηρετητή ώστε να καθοριστεί σε ποιον κόμβο θα αποθηκευτεί. Η ύπαρξη αυτού του σημείου ελέγχου αποτελεί ένα μοναδικό σημείο αποτυχίας για το συγκεκριμένο σύστημα και επομένως και ένα μεγάλο μειονέκτημα του συστήματος ONYX.

Θεωρούμε ότι η εξισορρόπηση του φόρτου των κόμβων σε ένα κατανεμημένο περιβάλλον μπορεί να είναι ζωτικής σημασίας για να επιτύχουμε υψηλές αποδόσεις καθώς και να σχεδιάσουμε ένα εύκολα επεκτάσιμο σύστημα. Ως αποτέλεσμα, στο πλαίσιο αυτής της διδακτορικής διατριβής προτείνουμε μια εναλλακτική αρχιτεκτονική για ένα σύστημα δημοσιεύσεων/συνδρομών που εκμεταλλεύεται την υψηλή απόδοση

των κατανεμημένων πινάκων κατακερματισμού (distributed hash tables). Οι κατανεμημένοι πίνακες κατακερματισμού αποτελούν μια ευρέως διαδεδομένη κατηγορία των δομημένων δικτύων ομότιμων κόμβων (structured peer-to-peer networks) και μέσω αυτής της αρχιτεκτονικής επιτυγχάνουμε να αποφύγουμε τις αδυναμίες των προηγούμενων προσεγγίσεων και να σχεδιάσουμε και αναπτύξουμε ένα πλήρως κατανεμημένο σύστημα όπου ο φόρτος κατανέμεται ισόποσα ανάμεσα στους κόμβους του δικτύου.

3. Σύστημα FoXtrot

Τόσο τα αυτόματα όσο και άλλες δενδρικές δομές έχουνε αποδειχθεί ιδιαιτέρων αποτελεσματικά στις πιο σημαντικές προτάσεις συστημάτων δημοσιεύσεων/συνδρομών XML δεδομένων όπως είναι το σύστημα YFilter και το σύστημα XPush. Η κύρια ιδέα της προσέγγισης που ακολουθούμε είναι ο σχεδιασμός μιας κατανεμημένης δενδρικής δομής σε ένα σύστημα κατανεμημένων πινάκων κατακερματισμού. Στο πλαίσιο αυτό, σχεδιάζουμε και προτείνουμε το σύστημα Foxtrot (**F**iltering **o**f **X**ML data in s**tr**uctured **o**verlay ne**t**works) για την διήθηση XML δεδομένων σε ένα κατανεμημένο περιβάλλον όπου οι κόμβοι είναι οργανωμένοι χρησιμοποιώντας ένα δομημένο δίκτυο ομοτίμων. Ο σχεδιασμός του συστήματος Foxtrot δεν απαιτεί την χρήση κάποιου σημείου κεντρικού ελέγχου (όπως για παράδειγμα απαιτεί το σύστημα ONYX) αλλά πρόκειται για ένα πλήρως κατανεμημένο σύστημα με υψηλή δυνατότητα κλιμάκωσης και ανοχής σε αποτυχίες κόμβων.

3.1 Κατανεμημένοι πίνακες κατακερματισμού

Ας περιγράψουμε αρχικά την αρχιτεκτονική πάνω στην οποία χτίζουμε το σύστημα FoXtrot και στην συνέχεια θα περιγράψουμε πως χρησιμοποιούμε τις βασικές λειτουργίες που μας προσφέρει για να «χτίσουμε» το προτεινόμενο σύστημα διήθησης XML δεδομένων. Οι κατανεμημένοι πίνακες κατακερματισμού στοχεύουν στην επίλυση του εξής βασικού προβλήματος αναζήτησης: «Έστω Χ ένα αντικείμενο δεδομένων αποθηκευμένο σε ένα κατανεμημένο δυναμικό δίκτυο από κόμβους. Βρες το αντικείμενο δεδομένων Χ.». Υπάρχουν διάφορες σχετικές προτάσεις όπως είναι το σύστημα Chord και το σύστημα Pastry αλλά σε όλες τις διαφορετικές υλοποιήσεις η βασική ιδέα είναι να λυθεί αυτό το πρόβλημα αναζήτησης προσφέροντας κάποια μορφή κατανεμημένης λειτουργικότητας μέσω ενός πίνακα κατακερματισμού. Θεωρώντας ότι αυτά τα αντικείμενα δεδομένων μπορούν να αναγνωριστούν χρησιμοποιώντας μοναδικά αριθμητικά κλειδιά, οι κόμβοι του δικτύου συνεργάζονται για να αποθηκεύσουν τα κλειδιά μεταξύ τους (τα αντικείμενα δεδομένων μπορεί να είναι τα πραγματικά δεδομένα ή δείκτες σε δεδομένα). Η απλή διεπαφή που παρέχεται από τους κατανεμημένους πίνακες κατακερματισμού αποτελείται από δύο βασικές λειτουργίες: (1) την λειτουργία put(*id*, *item*) που εισάγει ένα αντικείμενο με κλειδί *id* και τιμή item στο δίκτυου και (2) την λειτουργία get(*id*) που επιστρέφει έναν δείκτη στον κόμβο του δικτύου που είναι υπεύθυνος για το κλειδί *id*. Χρησιμοποιώντας ουσιαστικά αυτές τις βασικές λειτουργίες εκτελούμε τόσο την αποθήκευση των ερωτημάτων των χρηστών στο σύστημα FoXtrot όσο και την διήθηση των XML εγγράφων που δημοσιεύονται.

4. Δομικό ταίριασμα

Σε αυτήν την ενότητα περιγράφουμε την δομή που χρησιμοποιούμε για την διήθηση XML δεδομένων στο σύστημα FoXtrot δίνοντας αρχικά έμφαση στο «δομικό ταίριασμα».

4.1 Πεπερασμένα μη-ντετερμινιστικά αυτόματα

Επιλέξαμε να χρησιμοποιήσουμε για την ευρετηρίαση των ερωτημάτων των χρηστών, σε αντιστοιχία με το σύστημα YFilter, ένα πεπερασμένο μη-ντετερμινιστικό αυτόματο ((nondeterministic finite-state automaton ή NFA). Ένα μη ντετερμινιστικό πεπερασμένο

αυτόματο είναι ένα πεπερασμένο αυτόματο που από μία κατάσταση, διαβάζοντας ένα σύμβολο εισόδου, μεταβαίνει σε μία ή και παραπάνω καταστάσεις. Σκοπός μας είναι να κατασκευάσουμε, να αποθηκεύσουμε, και να εκτελέσουμε το αυτόματο που κωδικοποιεί ένα σύνολο ερωτημάτων XPath σε ένα κατανεμημένο περιβάλλον. Ο λόγος που επιλέγουμε να κατανέμουμε την συγκεκριμένη δομή στους κόμβους του δικτύου έχει να κάνει με την έμφυτη μη-ντετερμινιστική φύση αυτών των αυτομάτων που τους επιτρέπει να βρίσκονται σε πολλές καταστάσεις ταυτόχρονα. Αντιθέτως δεν επιλέγουμε να χρησιμοποιήσουμε ένα ντετερμινιστικό πεπερασμένο αυτόματο (DFA) όπου μπορούμε να μεταβούμε σε μία μόνο κατάσταση γιατί δεν επιτρέπει παραλληλία στην εκτέλεση του. Επισημαίνουμε πάντως ότι έχουνε προταθεί κεντρικοποιημένα συστήματα στην σχετική βιβλιογραφία που χρησιμοποιούνε τόσο μη-ντετερμινιστικά όσο και ντετερμινιστικά αυτόματα όπως είναι η πρόταση XPush.

4.2 Κατανομή NFA στο σύστημα FoXtrot

Η βασική ιδέα για την αποθήκευση του NFA αυτόματου στο σύστημα FoXtrot είναι ότι κάθε κόμβος του δικτύου είναι υπεύθυνος για ένα «κομμάτι» αυτής της δενδρικής δομής. Η κατανομή γίνεται στο επίπεδο των καταστάσεων του NFA (NFA states) καθορίζοντας για κάθε κατάσταση ένα κόμβο του δικτύου ο οποίος είναι «υπεύθυνος» για την συγκεκριμένη κατάσταση. Θεωρούμε επιπλέον την παράμετρο l, η οποία καθορίζει το μέγεθος του «κομματιού» για το οποίο είναι υπεύθυνος ο κάθε κόμβος. Αν η τιμή της παραμέτρου l είναι 0, τότε κάθε κατάσταση του NFA αποθηκεύεται μια μοναδική φορά στον έναν υπεύθυνο κόμβο. Για μεγαλύτερες τιμές της παραμέτρου l, κάθε κόμβος εκτός από την κατάσταση για την οποία είναι υπεύθυνος αποθηκεύει μαζί και τις επόμενες καταστάσεις που την «ακολουθούν» σε ένα μονοπάτι μήκους l. Με αυτόν τον τρόπο κάθε κατάσταση αποθηκεύεται σε περισσότερους από έναν κόμβους ή με άλλα λόγια οι κόμβοι αποθηκεύουν επικαλυπτόμενα «κομμάτια» του αυτομάτου, βελτιώνοντας με αυτόν τον τρόπο και την ανοχή του συστήματος σε αποτυχίες κόμβων. Χρησιμοποιούμε τον όρο «κατανεμημένο αυτόματο» ή «κατανεμημένο NFA» για να αναφερθούμε στο αυτόματο που μοιράζονται οι κόμβοι του συστήματος FoXtrot. Εκτός από την αντιγραφή καταστάσεων που προκύπτει από την χρήση της παραμέτρου l, χρησιμοποιούμε και άλλες τεχνικές αντιγραφής που στόχο έχουνε να εξασφαλίσουν εξισορρόπηση του φόρτου ανάμεσα στους κόμβους.

Για να καθορίσουμε ποιος κόμβος είναι υπεύθυνος για μια κατάσταση του NFA, αναγνωρίζουμε μοναδικά κάθε κατάσταση με ένα κλειδί. Με αυτόν τον τρόπο, χρησιμοποιώντας τις βασικές λειτουργίες ενός κατανεμημένου πίνακα κατακερματισμού που περιγράψαμε προηγουμένως, βρίσκεται ο κόμβος που θα είναι υπεύθυνος. Το κλειδί μιας κατάστασης δημιουργείται από τον συνδυασμό (concatenation) των ετικετών των μεταβάσεων (label transitions) του αυτόματου που οδηγούν σε αυτήν την κατάσταση. Τα ερωτήματα των χρηστών αποθηκεύονται μαζί με τις αντίστοιχες τελικές καταστάσεις (final ή accepting states) τους. Με άλλα λόγια, αν κατά την εκτέλεση του αυτομάτου φτάσουμε σε μια κατάσταση που αποτελεί τελική κατάσταση για ένα ερώτημα, σημαίνει ότι αυτό το ερώτημα έχει ικανοποιηθεί για την συγκεκριμένη είσοδο δεδομένων.

4.3 Κατασκευή κατανεμημένου NFA στο σύστημα FoXtrot

Αφού εξηγήσαμε πως ακριβώς θέλουμε να κατανεμηθεί το αυτόματα στους κόμβους του δικτύου, συνεχίζουμε περιγράφοντας πως «χτίζουμε» σταδιακά αυτό το αυτόματο καθώς εισάγονται νέα ερωτήματα στο σύστημα. Η βασική ιδέα για την προοδευτική κατασκευή του κατανεμημένου αυτομάτου είναι η ακόλουθη. Έστω ότι έρχεται ένα νέο ερώτημα και θέλουμε να το εισάγουμε στο κατανεμημένο αυτόματο του FoXtrot. Διασχίζουμε το αυτόματο ξεκινώντας από την αρχική κατάσταση και προχωρώντας από κατάσταση σε κατάσταση (ενδεχομένως χρειάζεται να επισκεπτούμε περισσότερους

από έναν κόμβους) μέχρι να βρεθεί είτε η τελική κατάσταση για αυτό το ερώτημα είτε να αναζητήσουμε μια κατάσταση η οποία δεν υπάρχει. Στην δεύτερη περίπτωση ο υπεύθυνος κόμβος δημιουργεί την συγκεκριμένη κατάσταση ενημερώνοντας ενδεχομένως και όλους τους κόμβους πρέπει να συμπεριλάβουν μια μετάβαση προς την νέα αυτήν κατάσταση. Με αυτόν τον τρόπο προσθέτουμε όσες μεταβάσεις και καταστάσεις χρειάζονται μέχρι να αποθηκευτεί επιτυχώς το συγκεκριμένο ερώτημα.

4.4 Εκτέλεση κατανεμημένου NFA στο σύστημα FoXtrot

Ολοκληρώνουμε την περιγραφή μας για το κατανεμημένο NFA του συστήματος FoXtrot περιγράφοντας πως ακριβώς εκτελούμε το αυτόματο όταν δημοσιεύονται νέα XML έγγραφα ώστε να ανακαλύψουμε τα ερωτήματα που ικανοποιούνται από αυτά και να ενημερώσουμε τους ενδιαφερόμενους χρήστες. Η διαδικασία που ακολουθούμε είναι αντίστοιχη με αυτήν που ακολουθείται στο κεντρικοποιημένο σύστημα YFilter όπου το XML έγγραφο χωρισμένο σε «συντακτικά συμβάντα» δίνεται ως είσοδος στο αυτόματο. Συγκεκριμένα χρειαζόμαστε μια στοίβα η οποία ανά πάσα στιγμή περιέχει όλες τις καταστάσεις που έχουμε ήδη επισκεφτεί και είναι ακόμη ενεργές και που μας επιτρέπει με αυτόν τον τρόπο να οπισθοδρομήσουμε (backtracking) όταν χρειαστεί. Για κάθε κατάσταση που βρίσκεται στην κορυφή της στοίβας και ονομάζεται «ενεργή κατάσταση» χρειάζεται να ανακτήσουμε τις επόμενες καταστάσεις που μπορεί να προκύψουν καθώς λαμβάνουμε ως είσοδο το επόμενο XML συμβάν. Η εκτέλεση τερματίζεται όταν αδειάσει η στοίβα εκτέλεσης. Δεδομένου ότι οι καταστάσεις είναι κατανεμημένες στους κόμβους του δικτύου, σε κάθε βήμα της εκτέλεσης πρέπει να προωθήσουμε το συγκεκριμένο συμβάν στους κόμβους που είναι υπεύθυνοι για τις «ενεργές καταστάσεις». Σχεδιάζουμε δύο μεθόδους για να επιτύχουμε αυτό, η πρώτη λειτουργεί επαναληπτικά ενώ η δεύτερη λειτουργεί αναδρομικά.

Η επαναληπτική προσέγγιση βασίζεται κυρίως στον κόμβο που είναι υπεύθυνος για την αρχική κατάσταση. Αυτός ο κόμβος δημιουργεί και διατηρεί την στοίβα για την εκτέλεση του αυτόματου και επιπλέον επικοινωνεί με όλους τους κόμβους των καταστάσεων που γίνονται ενεργές ώστε να ανακτήσει τις επόμενες καταστάσεις και να μπορέσει να ολοκληρωσεί την εκτέλεση του αυτομάτου. Η επαναληπτική προσέγγιση επιβαρύνει κυρίως τον κόμβο που αποθηκεύει την αρχική κατάσταση, ο οποίος είναι υπεύθυνος να επικοινωνεί με όλους τους άλλους κόμβους που έχουν καταστάσεις που απαιτούνται για την εκτέλεση και δεν είναι τοπικά αποθηκευμένες στον ίδιο τον κόμβο. Είναι αναμενόμενο, και το δείχνουμε επίσης μέσω πειραματικής αξιολόγησης, ότι η επαναληπτική προσέγγιση έχει μειωμένη απόδοση λόγω της συμφόρησης που δημιουργείται στον υπεύθυνο κόμβο.

Συνεχίζοντας με την αναδρομική προσέγγιση, παρατηρούμε ότι μπορεί να υπάρχουν πολλά ενεργά μονοπάτια κατά την εκτέλεση ενός NFA τα οποία είναι ανεξάρτητα και η εκτέλεση τους μπορεί να προχωρήσει παράλληλα από διαφορετικούς κόμβους. Σε αυτήν την περίπτωση ο κόμβος που λαμβάνει το XML έγγραφο το προωθεί πρώτα στον κόμβο που είναι υπεύθυνος για την αρχική κατάσταση ώστε να ξεκινήσει η εκτέλεση. Η εκτέλεση στην συνέχεια συνεχίζεται αναδρομικά με κάθε κόμβο υπεύθυνο για μια ενεργή κατάσταση να συνεχίζει την εκτέλεση στο δικό του μονοπάτι εκτέλεσης. Σε αυτήν την μέθοδο, η στοίβα εκτέλεσης δεν δημιουργείται ξεχωριστά αλλά εμπεριέχεται έμμεσα στις αναδρομικές εκτελέσεις των διαφορετικών μονοπατιών. Όπως δείχνουμε και πειραματικά ο φόρτος της εκτέλεσης με την αναδρομική μέθοδο μοιράζεται ανάμεσα σε περισσότερους κόμβους και επιπλέον η δυνατότητα παράλληλων εκτελέσεων οδηγεί την αναδρομική μέθοδο να έχει πολύ καλύτερη απόδοση σε σχέση με την επαναληπτική μέθοδο.

5. Ταίριασμα τιμών

Εκτός από το να χρησιμοποιήσουμε το κατανεμημένο αυτόματο για το δομικό ταίριασμα μεταξύ των ερωτημάτων και των δεδομένων, μελετούμε επιπλέον και το πρόβλημα ταιριάσματος τιμών (value predicates) που περιλαμβάνονται στα ερωτήματα και στα δεδομένα επιτυγχάνοντας ένα πιο επιλεκτικό φιλτράρισμα των δεδομένων. Ως ένα ενδεικτικό παράδειγμα ας θεωρήσουμε μια διαδρομή σε ένα XPath ερώτημα της μορφής «/βιβλιογραφία/άρθρο/αναφορά» που αναφέρεται σε άρθρα που περιλαμβάνουν κάποια αναφορά. Ας θεωρήσουμε τώρα ένα δεύτερο ερώτημα που επιπλέον περιλαμβάνει κάποιες τιμές που καθορίζουν το έτος δημοσίευσης του άρθρου ή το όνομα του συγγραφέα όπως είναι το ακόλουθο «/βιβλιογραφία / άρθρο [@ έτος > 2007] / συγγραφέας [text() = "Ίρις Μηλιαράκη"]). Ανάλογα με το πόσο επιλεκτικές είναι αυτές οι τιμές (για παράδειγμα θεωρήστε το πλήθος άρθρων ενός συγκεκριμένου συγγραφέα σε σχέση με το πλήθος άρθρων που έχουν δημοσιευτεί το 2007), ποικίλει η επιλεκτικότητα του ερωτήματος και το ποσοστό των XML εγγράφων που μπορεί να ταιριάζει. Με άλλα λόγια ο αριθμός των ερωτημάτων τα οποία είναι ταιριάζουν μόνο δομικά μπορεί να είναι μεγάλος οδηγώντας σε μια σπατάλη των σχετικών πόρων όταν το ταίριασμο είναι μόνο δομικό. Λαμβάνοντας υπόψιν τις προηγούμενες παρατηρήσεις, σχεδιάζουμε για το σύστημα FoXtrot τεχνικές που συνδυάζουν τόσο το δομικό ταίριασμα όσο και των ταίριασμα των τιμών μεταξύ ερωτημάτων και δεδομένων σε ένα πλήρως κατανεμημένο περιβάλλον. Ο συνδυασμός του δομικού ταιριάσματος με το ταίριασμα των τιμών μπορεί να γίνει με διάφορους τρόπους και στην συνέχεια περιγράφουμε τις διαφορετικές προσεγγίσεις που μπορούμε να ακολουθήσουμε.

5.1 Μέθοδος «top-down»

Η πρώτη μέθοδος ελέγχει τις τιμές αφού ολοκληρωθεί το δομικό ταίριασμα και επομένως επεξεργάζεται τα έγγραφα XML «από πάνω προς τα κάτω» (top-down) προχωρώντας από την ρίζα προς τα φύλλα. Άρα, χρησιμοποιούμε αρχικά το κατανεμημένο NFA για να υπολογίσουμε το αρχικό σύνολο των ερωτημάτων που ταιριάζουν δομικά με το XML έγγραφο. Έπειτα γίνεται αποτίμηση των τιμών και υπολογίζουμε το τελικό σύνολο ερωτημάτων που ικανοποιείται συνολικά από το XML έγγραφο.

Με άλλα λόγια αυτή η μέθοδος ελέγχει τις τιμές αφού ολοκληρωθεί η εκτέλεση του αυτόματου και την ονομάζουμε μέθοδο «top-down». Δεδομένου ότι στο σύστημα FoXtrot η εκτέλεση του αυτόματου γίνεται παράλληλα από διάφορους κόμβους, ο κάθε ένας από αυτούς τους κόμβους ανακαλύπτει ένα υποσύνολο ερωτημάτων που έχουνε ταιριαστεί με βάση την δομή τους. Μόλις ένας κόμβος ανακαλύπτει αυτά τα ερωτήματα, είναι υπεύθυνος να ελέγξει και τις τιμές.

5.1 Μέθοδος «bottom-up»

Για την δεύτερη μέθοδο εμπευστήκαμε από μια ευρέως διαδεδομένη στρατηγική στην περιοχή της βελτιστοποίησης σχεσιακών βάσεων δεδομένων όπου οι πράξεις επιλογής (selections) εφαρμόζονται όσο το δυνατόν νωρίτερα κατά την εκτέλεση μιας επερώτησης. Εφαρμόζοντας λοιπόν το ίδιο, ελέγχουμε πρώτα τις τιμές των ερωτημάτων και στην συνέχεια ελέγχουμε την δομή. Σε αντίθεση με την προηγούμενη μέθοδο, επεξεργαζόμαστε τα XML έγγραφα από τα φύλλα προς την ρίζα (δεδομένου ότι οι τιμές σε μια XML δενδρική δομή βρίσκονται πάντα στους κόμβους-φύλλα) και επομένως ονομάζουμε αυτήν την μέθοδο «bottom-up».

Σε αντίθεση με τις άλλες μεθόδους για να επιτύχουμε την μέθοδο «bottom-up» πρέπει να αλλάξουμε τον τρόπο που μοιράζουμε τα ερωτήματα στους κόμβους του δικτύου. Ο στόχος είναι να μπορούμε να ανακαλύψουμε εύκολα και γρήγορα τα ερωτήματα που

περιλαμβάνουν κάποιες συγκεκριμένες τιμές. Για αυτόν τον λόγο κατανέμουμε τα ερωτήματα στους κόμβους βάσει των τιμών που περιέχουν.

Ένα μειονέκτημα της μεθόδου «bottom-up» είναι ότι το ευρετικό του να εκτελέσουμε πρώτα τις πράξεις επιλογής μπορεί να λειτουργεί εξαιρετικά στις βάσεις δεδομένων αλλά στην περίπτωση του FoXtrot, οι κόμβοι μπορεί να σπαταλήσουν πόρους στο ταίριασμα των τιμών ερωτημάτων των οποίων η δομή δεν ταιριάζει με το XML έγγραφο.

5.1 Μέθοδος «step-by-step»

Η τελευταία μέθοδος ονομάζεται «step-by-step» και λαμβάνει υπόψιν ότι τα ερωτήματα XPath αποτελούνται από διακριτά βήματα και καθένα από αυτά τα βήματα μπορεί να περιλαμβάνει ένα ή περισσότερους περιορισμούς τιμών. Επομένως, η μέθοδος «step-by-step» επιτρέπει να ελέγχονται οι τιμές κατά την διάρκεια της εκτέλεσης του αυτομάτου (ή αλλιώς κατά την διάρκεια του δομικού ταιριάσματος) βήμα προς βήμα.

6. Πειραματική αξιολόγηση

Η πειραματική αξιολόγηση του συστήματος FoXtrot έχει γίνει σε δύο διαφορετικά περιβάλλοντα. Αρχικά επιλέξαμε μια διαθέσιμη τοπική συστάδα υπολογιστών (machine cluster), στοχεύοντας να μελετήσουμε την απόδοση των διαφορετικών τεχνικών του συστήματος μας σε ένα ελεγχόμενο περιβάλλον εκμηδενίζοντας με αυτόν τον τρόπο τους εξωγενείς παράγοντες. Έπειτα αξιολογήσαμε το σύστημα μας στο δίκτυο PlanetLab, το οποίο αποτελεί μία ευρέως διαδεδομένη πλατφόρμα που προσομοιώνει σε ικανοποιητικό βαθμό τις πραγματικές συνθήκες του διαδικτύου. Σε ένα τέτοιο περιβάλλον επομένως, κόμβοι μπορεί να αποτύχουν καθώς και νέοι κόμβοι να εισέλθουν στο σύστημα άνα πάσα στιγμή.

Δείχνουμε ότι μεταβάλλοντας το μέγεθος του αυτόματου που διατηρεί ο κάθε κόμβος (μέσω της παραμέτρου l) μπορούμε να επιτύχουμε διαφορετικά αποτελέσματα καθώς και διαφορετική διανομή του φόρτου. Μελετάμε επίσης πως τα διάφορα χαρακτηριστικά, τόσο των ερωτημάτων όσο και των XML δεδομένων (όπως το βάθος των ερωτημάτων ή το σύνολο τιμών που περιλαμβάνουν), επηρεάζουν την απόδοση του συστήματος. Μέσω των πειραμάτων που διεξάγαμε επιδείξαμε την υπεροχή του συστήματος FoXtrot το οποίο μπορεί να αποθηκεύσει εκατομμύρια ερωτήματα χρηστών επιτυγχάνοντας ταυτόχρονα υψηλές αποδόσεις φιλτραρίσματος XML δεδομένων (συγκεκριμένα της τάξης των 1000 ερωτημάτων άνα δευτερόλεπτο). Η ταχύτητα διανομής των συγκεκριμένων ειδοποιήσεων προς τους ενδιαφερόμενους χρήστες έφτασε τις 1500 κοινοποιήσεις ανά δευτερόλεπτο. Μελετήσαμε επιπλέον την δυνατότητα κλιμάκωσης του συστήματος μας αυξάνοντας διαδοχικά το μέγεθος του δικτύου και ελέγχοντας πως αυτό επηρεάζει την απόδοση του.

7. Συμπεράσματα

Στα πλαίσια αυτής της διδακτορικής διατριβής περιγράφουμε το σύστημα FoXtrot. Το σύστημα FoXtrot είναι ένα πλήρως κατανεμημένο σύστημα διήθησης XML δεδομένων που χρησιμοποιεί αρχιτεκτονική των κατανεμημένων πινάκων κατακερματισμού. Συνδυάζουμε την υψηλή απόδοση ενός αυτομάτου NFA για το ταίριασμα ερωτημάτων XPath και δεδομένων XML και την ευελιξία των κατανεμημένων πινάκων κατακερματισμού για το σχεδιασμό ενός κλιμακώσιμου συστήματος. Επικεντρωνόμαστε στον σχεδιασμό μιας προσέγγισης που κατανέμει τον φόρτο ισοδύναμα στους κόμβους του δικτύου αποφεύγοντας τα σημεία συμφόρησης που μπορούν να μειώσουν την απόδοση του συστήματος.

Σχεδιάζουμε και συγκρίνουμε διαφορετικές μεθόδους τόσο για το δομικό ταίριασμα XML δεδομένων όσο και για ταίριασμα τιμών. Με μια εκτενή πειραματική αξιολόγηση

δείχνουμε ότι το σύστημα FoXtrot μπορεί να αποθηκεύσει εκατομμύρια ερωτημάτων επιτυγχάνοντας ταυτόχρονα υψηλή απόδοση φιλτραρίσματος XML δεδομένων. Δείχνουμε επιπλέον ότι η απόδοση του συστήματος βελτιώνεται καθώς αυξάνουμε το μέγεθος του δικτύου εκτελώντας πειράματα τόσο σε ένα ελεγχόμενο περιβάλλον εκμηδενίζοντας τους εξωγενείς παράγοντες όσο και σε ένα ιδιαίτερα δυναμικό δίκτυο που παρέχεται από την πλατφόρμα PlanetLab.

7.1 Μελλοντικές επεκτάσεις

Ως μελλοντικές επεκτάσεις θέλουμε να μελετήσουμε επεκτάσεις του μοντέλου ερωτήσεων που χρησιμοποιούμε ώστε να υποστηρίξουμε την πλήρη εκφραστικότητα της γλώσσας XPath. Επιπλέον είναι ενδιαφέρον να δούμε κατά πόσο οι τεχνικές μας μπορούν να εφαρμοστούν και για πιο εκφραστικά μοντέλα δεδομένων όπως είναι το μοντέλο δεδομένων RDF (Resource Description Framework) που είναι ευρέως διαδεδομένο στην περιοχή του Σημασιολογικού Ιστού (Semantic Web).

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

As the Web is growing continuously, a great amount of data becomes available to more and more users, making it difficult for them to discover interesting information by searching. Users cannot cope with this information explosion and while attempting to stay informed by browsing through a vast amount of new information, they fail to take advantage of the dynamic nature of the Web. As successfully observed by Mitch Kapor:

"Getting information off the Internet is like taking a drink from a fire hydrant"

Mitch Kapor, founder of Lotus Development Corporation

In the client/server communication model typically found in large-scale distributed systems, the communication between clients and servers is synchronous causing the client to be blocked from the time it issues a request until it receives a reply. This kind of synchronous communication complicates the development and maintenance of dynamic large-scale Internet applications. If we also consider the continuous increase in the scale of available distributed systems, it becomes evident that a more flexible and less tightly coupled communication model is required. The key idea towards designing such a model lies in achieving loose coupling between the producers and the consumers of data. In other words, producers should be able to publish information into the network without knowing the identity, the location or the total number of consumers. Likewise, consumers should be able to express their interests without knowing which data producers can provide relevant content.

Eugster et al. [31] analyzed this kind of *decoupling* along the following three dimensions: *synchronization*, *time* and *space*. Synchronization decoupling allows consumers and producers to communicate in an asynchronous way. As a result, data consumers are not blocked while waiting for data to arrive and likewise data producers are not blocked waiting for consumers to receive their data. Time decoupling allows producers and consumers to exchange data without needing to be active at the same time. Finally, space decoupling allows consumers and producers to remain anonymous to each other.

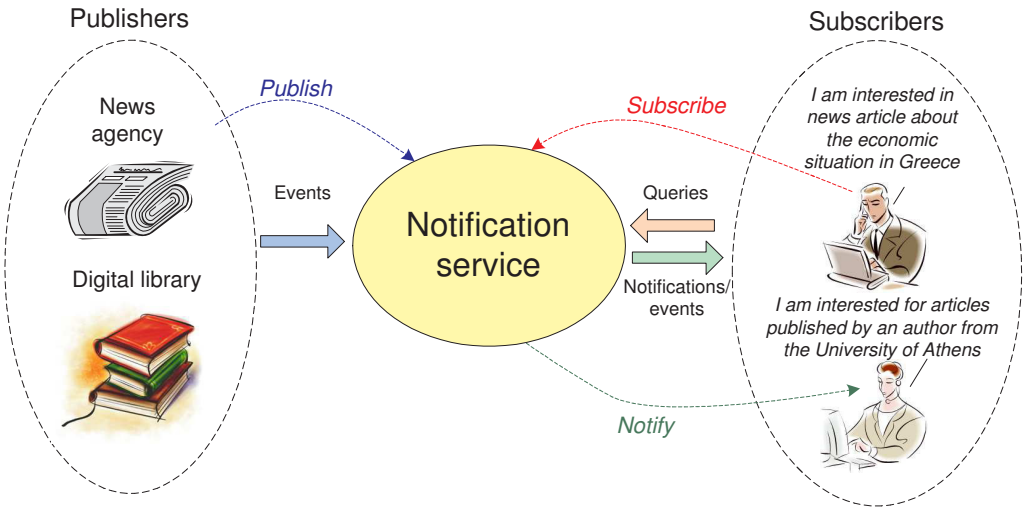


Figure 1.1: Overview of a publish/subscribe system

The *publish/subcribe paradigm* unifies all the above types of decoupling and has received a lot of attention in the literature. It provides a flexible and loosely coupled form of communication enabling selective dissemination of information.

1.1 Publish/subscribe paradigm

A high-level architecture of a publish/subscribe system is depicted in Figure 1.1. As the figure illustrates, a typical publish/subscribe system involves the following parties:

- The *subscribers* who express their interests by submitting a *subscription* and waiting to be notified whenever an event of interest is published. Subscribers are also called *consumers* since they consume events.

- The *publishers* who publish information and generate *events*. Publishers are also called *producers* since they produce events.

- The *notification service* which is the basic infrastructure for managing the subscriptions of the consumers and delivering them relevant events from the publishers. The notification service realizes the interaction between publishers and subscribers and allows them to act independently of each other as we described above.

Before proceeding with surveying different subscription and publication models proposed for publish/subscribe systems and describing different architectures for building a notification service, we present some interesting applications.

1.1.1 Applications

Applications of publish/subscribe systems include various popular notification services such as news monitoring, blog monitoring, product monitoring (e.g., for price changes) and alerting services for digital libraries. For the case of digital libraries, Elsevier alerting services[1] is a interesting example since it allows users to subscribe for specific topics, search results or even citations of a specific publication. Another popular service that has emerged for everyday use is *Google alerts*[2]. This is a notification service offered by Google that provides email alerts to users about the latest Google results. Users can either select a topic or specify a list of keywords and monitor a developing news story or get the latest on their favorite sports teams. Other popular services in the area of e-commerce include alert services about price drops and hot deals for products like the *BuyLater* website[3].

1.1.2 Models and languages

Subscribers are typically interested in only a subset of the events produced by various providers. Depending on the data model and language used to express publications and subscriptions different levels of expressiveness can be offered. Such a decision is of great importance since it affects the overall capabilities of the system. Several subscription models have been proposed in the literature for expressing the interests of subscribers and can be classified in two major categories, namely *topic-based* or *subject-based* and *content-based* [31].

In topic-based publish/subscribe systems, events are assigned to *topics*. Consumers can select to subscribe to one or more topics, typically identified by keywords, and will then receive all relevant events that belong to these topics. Some of the earliest topic-based systems include SCRIBE [19] and Bayeux [113]. A useful enhancement of the topic-based subscription model is to use an hierarchy for topics. In this case, a subscription to a node of this hierarchy also involves subscribing to all the subtopics of that nodes. As shown in the literature, topic-based systems are easier to implement than content-based ones due to the simplicity of their data model. However, this is also their disadvantage: their subscription model is static and allows only a limited level of expressiveness.

Content-based publish/subscribe systems employ a more expressive subscription model by allowing subscriptions to be constructed using the actual content of the events. The generality of this model is the reason it has received a lot of interest in the literature. In this context, many systems have been built using different data models like the attribute-value

[1]Elsevier alerting services: `http://www.elsevier.com/alerts`
[2]Google alerts (beta): `http://www.google.com/alerts`
[3]Buy later: `http://www.buylatr.com`

model [18, 100, 6, 42], XML data model [7, 16, 40, 39, 41, 56, 30, 20] and RDF data model [78]. Other well-known content-based systems include Gryphon [45, 3] and LeSubscribe [76].

Since the content-based approach allows highly expressive data models, it also requires more complex protocols and implemented systems can suffer from higher overheads. For example, the more expressive a subscription model is, the more complex the algorithms required to manage and filter subscriptions are.

1.1.3 The filtering problem

In a publish/subscribe system, the notification service has to solve the following important problem:

> Given a set of subscriptions S and a publication p, identify all the subscriptions $s \in S$ that match publication p.

Depending on the publication data model and the subscription language used, the semantics of matching and the algorithm for solving this can differ substantially. In the literature, different semantics considered for matching include topic-based, predicate-based and XML-based semantics. Given a semantics for matching, the algorithm for solving the filtering problem goes hand-in-hand with the choice of architecture for the notification service. The possibilities for this choice are discussed immediately.

1.1.4 Architectures

In the literature, the following classes of system architecture have been proposed for building the notification service: *centralized systems*, *content-based overlay networks* and *peer-to-peer networks*.

A centralized architecture is the easiest to implement since the notification service consists of a single component performing all relevant tasks. This component is typically referred to as a *broker* or a *server*. Centralized solutions suffer from well-known weakness including lack of scalability and zero fault-tolerance, and are not considered suitable for realizing a large-scale publish/subscribe system.

Content-based overlay networks consist of a collection of routers responsible for forwarding data towards interested subscribers. Content-based routers, also called brokers, route data based on their content and are usually organized in mesh or tree-based configurations. The tasks performed by the notification service are distributed among these routers. This approach can achieve scalability as the size of the system grows depending on the protocols utilized.

Finally, peer-to-peer (P2P) networks are distributed systems consisting of a very large number of computing nodes that cooperate for sharing data without any centralized control. They go beyond the typical client-server systems since each peer can play both the role of a client and a server. Some of their most desirable features include robustness, efficient search of data items, anonymity, massive scalability and fault tolerance. Because of these properties, several publish/subscribe systems have been implemented over P2P networks [113, 96, 99].

An important issue affecting the performance of distributed publish/subscribe systems based on content-based overlay networks of or P2P networks is *load balancing*. A network is load-balanced if all its nodes are sharing the tasks required to run the notification service functionality as fairly as possible. These tasks include managing and storing subscriptions, performing matching, forwarding and delivering events.

1.2 Fundamental questions

To design and implement a large-scale publish/subscribe system, we need to address a number of interesting challenges as we discussed above. In this thesis, we look into the following fundamental questions underlying the design and implementation of such a system:

- Which subscription model is appropriate in the Web era and allows an adequate level of expressiveness for consumers to express a plethora of diverse interests?
- Which architecture is appropriate for offering publish/subscribe functionality on an Internet-scale facilitating scalability and reliability?
- Which algorithms to employ for performing filtering and manage subscriptions achieving good performance using the aforementioned architecture?

In the rest of this thesis we provide answers to the above questions and present our original results for advancing the state of the art in the above problems. For achieving this, we design, develop and evaluate an distributed XML filtering system and demonstrate how we meet these challenges.

1.3 Solution outline

Let us provide now an overview of our answers to the questions posed earlier and argue that our approach provides solutions that outperform the relevant literature.

1.3.1 XML as a subscription model

Over the past decade, eXtensible Markup Language XML [15] has been widely used for data representation and exchange on the Web. The properties of XML, including simplicity, flexibility and extensibility, have assisted in becoming a de facto standard on the Internet and also make XML a good fit for our purposes. It also suitable since it separates data and presentation enabling heterogeneous systems to exchange data without knowing how this data is represented internally. We are not alone in this choice. Due to the popularity and advantages of the XML data model, a lot of research has focused on designing efficient and scalable XML filtering systems [30, 20, 41, 16, 98].

In XML-based publish/subscribe systems, subscribers submit continuous queries expressed in XPath/XQuery asking to be notified whenever their queries are satisfied by incoming XML documents. As an example, consider XPath query `article[author=` *"John Smith"* `]`, where a user wants to be notified when author John Smith publishes an article. Users can specify in XPath queries constraints over both the structure and values of XML documents enabling a more precise filtering of the XML documents.

1.3.2 Distributed hash tables as the architecture

Many approaches have been proposed in recent years like YFilter [30] and its predecessor XFilter [7], XTrie [20], IndexFilter [16] and many others [41, 39, 44, 73] performing XML filtering in a centralized environment. As a result, these proposals suffer from the typical problems of centralized solutions, such as limited scalability, single point of failure, network bottlenecks and zero fault tolerance. To overcome these weaknesses and offer XML filtering functionality on Internet-scale, such a service should be deployed in a distributed environment.

Earlier proposals like XNet [22], ONYX [29], parallel and hierarchical XTrie [33] and others [92, 24, 38, 102, 21] have also proposed to implement XML filtering in a distributed way. With respect to the distributed setting supported by the former techniques, the majority of these approaches assume a content-based overlay network with brokers responsible for forwarding XML data towards interested subscribers. Brokers route XML data based on their content and can be organized in meshes [92] or tree-based configurations [33, 29, 22]. Depending on design decisions that are specific to each system, these proposals do not exhibit good load balancing properties. Thus a performance deterioration can easily occur especially as the number of subscriptions increases, incoming events arrive at a high rate, or a large number of notifications is generated.

In addition, these architectures have not been designed so that they can cope with failures in a graceful way. This is an important disadvantage since large-scale XML filtering systems

run on top of the Internet we expect them to continue operating in the presence of failures.

We deal with the disadvantages of earlier systems by proposing an alternative architecture that exploits the power of distributed hash tables (DHTs), a well-known class of structured overlay networks, overcoming the weaknesses of the earlier and developing a fully distributed load balanced system exhibiting resilience to failures.

1.3.3 Scalable and efficient filtering algorithms

Considering that the amount of subscriptions we want to evaluate is very large, evaluating one query at a time in a brute force way becomes prohibitively expensive. The main challenge in building a notification service that provides efficient XML filtering against a large set of subscriptions is to effectively organize these subscriptions. A popular method to accomplish this is by designing and building some kind of index over the queries and perform filtering of the XML documents against a much smaller set of queries.

Works like YFilter [30], XTrie [33], XPush [41] and many others employ an XML filtering strategy based on automata or tree-based structures. While these strategies have been used with success for representing a set of queries and identifying XML documents that *structurally match* XPath queries, little attention has been paid to *value matching* (i.e., evaluation of value-based predicates) and especially in distributed environments. This can become an important problem since typical queries, apart from defining a structural path (e.g., `/dblp/article/author`), also contain value-based predicates (e.g., `/dblp/article[@year` $>$ `2007]`, `/author[text() =`"*Iris Miliaraki*"`]`) and depending on the selectivity of these predicates, the number of queries which are only structurally matched (i.e., false positives), might be large. For this reason, the benefit of using a filtering engine for structural matching, can be diminished. In other words, the XML filtering service should be able to scale with respect to both the number of the queries indexed and the predicates included in the queries.

The main idea of the approach studied in this thesis is to combine techniques for structural and value XML filtering in a distributed environment achieving high performance and scalability for a large set of applications.

1.3.4 Contributions

The main contributions of this thesis can be described in more detail as follows:

- We design, implement and evaluate a fully-distributed system called FoXtrot (***F****iltering* ***o****f* ***X****ML data on top of s****tr****uctured* ***o****verlay ne****t****works*) for efficient filtering of XML data on very large sets of XPath queries To achieve this, we show how a nondeterministic

automaton can be distributed among the nodes of a DHT and be executed by methods that exploit the inherent parallelism of an NFA. This *distributed* NFA is maintained by having peers being responsible for overlapping fragments of the corresponding NFA and different peers participating in the filtering process by executing in parallel several paths of the NFA. The size of these fragments is a tunable system parameter that allows us to control the amount of generated network traffic and load imposed on each peer. In addition, our distribution scheme adds redundancy to our system and as a result it also increases its fault tolerance.

- We show that our approach overcomes the weaknesses of typical content-based XML dissemination systems built on top of meshes or tree-based overlays with respect to load balancing. The design of FoXtrot allows us to employ simple yet effective replication methods for achieving a balanced load distribution. In addition, there is no need for a centralized component for assigning queries to network peers, since queries are distributed among the peers using the underlying DHT infrastructure.

- We demonstrate that apart from structural matching, our system FoXtrot can also deal in an efficient way with value-based predicates. To the best of our knowledge, our work is the first to study combined structural and value matching of queries in a distributed setting emphasizing on both operations. We describe, compare and implement different methods for combining structural and value XML filtering in FoXtrot.

- We perform an extensive experimental evaluation of FoXtrot in both the controlled environment provided by a local cluster and PlanetLab to exhibit the performance of our methods. We demonstrate that FoXtrot can index millions of user queries, achieving a high throughput of around 1000 queries per second in the local cluster. With respect to filtering, FoXtrot generates and disseminates more than 1500 notifications per second for the filtering scenarios we consider. We also show that our system exhibits scalability with respect to the network size, improving its performance as we add more peers to the network.

Since any path query can be transformed into a regular expression and consequently there exists an NFA for representing this query, our techniques described using XML and XPath can be applied to other data models and query languages. For example, Pérez et al. [77] show how to construct an NFA for indexing RDF path queries and provide us with the basis to apply our results to RDF.

1.4 Published papers

The results of this thesis have been published in the following papers ([66] and [67]):

- I. Miliaraki, Z. Kaoudi and M. Koubarakis. *XML Data Dissemination using Automata on top of Structured Overlay Networks*. 17th International World Wide Web Conference (WWW 2008), Beijing, China, 2008.
- I. Miliaraki and M. Koubarakis. *Distributed Structural and Value XML Filtering*. 4th ACM International Conference on Distributed Event-Based Systems (DEBS 2010), Cambridge, United Kingdom, 2010.

Also the following journal version of our research has been submitted to a major international journal [68]:

- I. Miliaraki and M. Koubarakis, *FoXtrot: Distributed Structural and Value XML Filtering*, Submitted to a journal.

1.5 Thesis structure

The remainder of this thesis presents related work, the FoXtrot system, its core techniques for combined structural and value matching, and results of an extensive experimental evaluation of the techniques fully implemented in the system. Chapter 2 provides background on the technical context in which the research of this thesis is conducted by describing the XML data model, the XPath query language, the technology of peer-to-peer networks an providing some background knowledge about finite automata. It also covers related research in this area. In Chapter 3, we present the architecture of our XML filtering system along with its data and query models. Chapter 4 describes our basic filtering techniques for structural matching in FoXtrot and also includes their experimental evaluation. Next, in Chapter 5, we focus on value matching techniques, discuss a number of different methods and also include their evaluation. Chapter 6 concludes this thesis by summarizing our contributions and discussing directions for future research.

Chapter 2

Background and related research

In this chapter, we provide background information required to understand the rest of this thesis. We first present the XML data model and describe how XML documents are structured, define the classes of well-formed and valid XML documents, and present the XPath query language for querying XML data. Second, we describe finite automata, give formal definitions of both deterministic and nondeterministic ones, and also give some example applications. Third, we present a survey of peer-to-peer networks focusing on structured overlays. We conclude this chapter, providing a detailed survey of previous research relevant to the topics of this thesis ranging from centralized and distributed XML filtering methods to works that distribute tree-like or other hierarchical structures on top of distributed hash tables.

2.1 Extensible Markup Language (XML)

The XML language [15] has been widely used as the standard format for data exchange on the Web and other environments. It uses a simple and very flexible text format derived from SGML[1]. In this section, we first describe the XML data model and how XML documents are structured. Then, we give a brief description of schema languages for specifying the structure of XML documents. Last, we introduce the XML query language XPath.

2.1.1 XML documents

We introduce the hierarchical data model used in XML language. An XML document can be represented using a rooted, ordered, labeled tree where each node represents an element or a value and each edge represents relationships between nodes such as an element - subelement

[1]Standard Generalized Markup Language was developed and standardized by the International Organization for Standards (ISO) in 1986

```
<dblp>
   <inproceedings key="conf/www/MiliarakiKK08" mdate="2008-05-13">
      <author>Iris Miliaraki</author>
      <author>Zoi Kaoudi</author>
      <author>Manolis Koubarakis</author>
      <title> XML data dissemination using ... overlay networks.</title>
      <pages>865-874</pages>
      <year>2008</year>
      <booktitle>WWW</booktitle>
      <url>db/conf/www/www2008.html#MiliarakiKK08</url>
   </inproceedings>
</dblp>
```

Figure 2.1: An example XML document from DBLP XML records

relationship. There is no limit on the nesting level of elements. Element nodes may also contain attributes, which describe their additional properties, or textual data. With respect to syntax, elements are identified by their start-tag and end-tag. The tag names are enclosed between angled brackets (i.e., `<tag name>`), while for end-tags an additional backslash is used (i.e., `</tag name>`). The content of an element is included between its start- and end- tags, and can contain textual data and sub-elements. The start-tag may also include a list of attributes.

We define the following characteristics for an XML document. The *depth* of an XML document is the longest nesting of an element appearing in the document. The *fanout* of an XML document is the maximum number of children nodes emerging from an element node.

Example 2.1.1 (An XML document)**.** *Figure 2.1 shows the textual representation of an example XML document that describes a conference publication as found in the DBLP bibliographic database*[2]*, while Figure 2.2 illustrates its corresponding tree structure. The root element of this document is* **<dblp>***, which refers to the DBLP bibliographic database. The child node of* **<dblp>** *is* **<inproceedings>** *element which represents a conference publication. The* **<inproceedings>** *element has two attributes, namely* **key** *and* **mdate** *with values* "conf/www/MiliarakiKK08" *and* "2008-05-13"*, respectively. Attribute* **key** *is used as a unique record identifier, while attribute* **mdate** *refers to the date this record was last modified. Element* **<title>** *includes as textual data the title of the publication* "XML data dissemination using automata on top of structured overlay networks"*. The element* **<booktitle>***, referring to the conference this paper was published, has no associated attributes. The depth of this XML document is* 2 *and its fanout is* 8 *(which corresponds to the children nodes of* **<inproceedings>** *element). Throughout this dissertation, we will use as our main example the DBLP XML records [1], which correspond to a well-understood paradigm.*

[2]DBLP Computer Science Bibliography: `http://dblp.uni-trier.de`

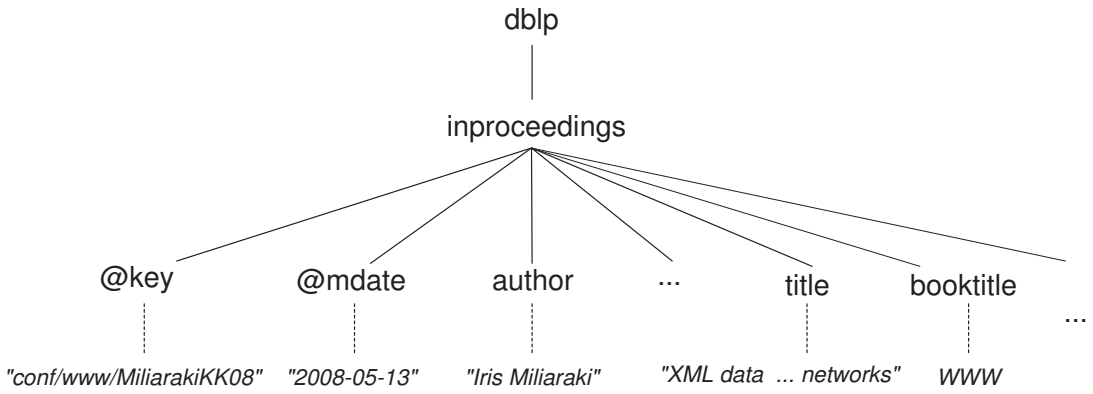


Figure 2.2: The XML tree of a DBLP XML record

With respect to the correctness of XML documents, we distinguish between *well-formed* and *valid* XML documents. An XML document is *well-formed* if it follows certain syntax rules defined in XML 1.0 specification [15]. Particular, it must start with an XML declaration indicating the version of XML being used and some other relevant attributes. An example declaration is the following which specifies the XML version and the character encoding supported:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

A well-formed document must also follow the syntactic guidelines of the tree model. There is exactly a single element, called the *root* or document element, no part of which appears in the content of any other element [15]. For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. In other words, the elements as specified by their respective start- and end-tags, nest properly within each other.

Apart from well-formedness, a stricter criterion for XML documents is validity. A well-formed XML document is *valid* if there is a document type definition (DTD) [15] or an XML schema [97] associated with it and the document complies with it. A document type definition specifies a general set of rules including all possible structures of the XML document and the domains of the attributes. DTD files are specified using a special syntax demonstrated in the following example.

Example 2.1.2 (An XML DTD file)**.** *A part of the DBLP DTD*[3] *is shown in Figure 2.3. First, a name is specified for the root element of the document which is called* `dblp`*. The XML root element* `<dblp>` *contains a long sequence of various bibliographic records. The DTD lists the different available elements that can be used as bibliographic records (namely* `article`*,* `inproceedings`*,* `proceedings`*,* `book`*,* `incollection`*,* `phdthesis`*,* `mastersthesis` *and* `www`

[3]DBLP DTD file as found in `http://dblp.uni-trier.de/xml/dblp.dtd`

```
<!-- DBLP DTD -->
<!ELEMENT dblp (article|inproceedings|proceedings|book|incollection|
                phdthesis|mastersthesis|www)*>
<!ENTITY % field "author|editor|title|booktitle|pages|year|address|
                  journal|volume|number|month|url|ee|cdrom|cite|
                  publisher|note|crossref|isbn|series|school|chapter">
<!ELEMENT inproceedings (%field;)*>
<!ATTLIST inproceedings key CDATA #REQUIRED
                        mdate CDATA #IMPLIED
>
...
```

Figure 2.3: A part of the DBLP DTD file

elements). For element ***<inproceedings>****, it defines a list of attributes containing attribute* ***key*** *as a required attribute and attribute* ***mdate*** *as an optional one.*

We can see that our example XML document depicted in Figures 2.1 and 2.2 conforms to the XML DTD of Figure 2.3. To require that an XML document conforms to a DTD, we specify this in the declaration of the document. The first lines of the example document should be changed as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE bib SYSTEM "dblp.dtd">
```

Even though XML DTD is considered adequate for specifying the tree structure of XML documents, it has several limitations. The main limitation is that it has its own specialized syntax. As a result, a different parser is required for processing it. These drawbacks led to the development of the XML schema language [97] which has been used as a standard for specifying the structure of XML documents using the syntax rules of XML itself. XML Schema provides a superset of the capabilities found in DTDs.

Note that the techniques described in this thesis for XML document filtering do not require XML documents to conform to a DTD or an XML schema but could exploit such schema information, if present, to improve their performance.

2.1.2 XML Path Language

Having described the XML data model and how XML documents are structured, we now proceed to languages used for querying XML data. There have been many proposals for XML query languages but two standards have emerged. The first is XML Path Language (XPath) [25] which models an XML document as a tree of nodes. XQuery language [13] is the second corresponding to a more general query language that uses XPath expressions but also

has additional constructs. In this section, we describe in detail XPath which is the language we support in this thesis. Our description is based on the relevant W3C specification [25].

2.1.2.1 Location paths

XPath [25], as its name indicates, is a language for navigating through the tree structure of an XML document. As described, there are different types of tree nodes, including element nodes, attribute nodes and text nodes. XPath treats an XML document as a tree and offers a way to select paths of this tree. Each XPath expression uses a *location path*. A location path consists of a sequence (one or more) *location steps* separated using a child (`/`) or a descendant (`//`) axis. The steps in a location path are composed together from left to right. There are two kinds of location paths, *relative location paths* and *absolute location paths*. An absolute location path starts with a child axis (`/`) which by itself selects the root node of the document. Otherwise, the location path is called relative starting with a node called the *context node*. Let us now study the structure of a single location step. A location step has three parts:

1. an *axis*, which specifies the tree relationship between the nodes selected by the location step and the context node,
2. a *node test*, which specifies the node type and node name selected by the location step, and
3. zero or more *predicates*, which use arbitrary expressions to further refine the set of nodes selected by the location step.

2.1.2.2 Axes, nodes tests and predicates

With respect to the first part of a location step which is an *axis*, XPath query language supports the following axes:

- The `self` axis contains the context node.
- The `child` axis contains the children of the context node.
- The `descendant` axis contains the descendants of the context node where a descendant is a child or a child of a child and so on. The `descendant` axis never contains attribute or namespace nodes.
- The `descendant-or-self` axis contains the context node and the descendants of the context node.

- The `parent` axis contains the parent of the context node.
- The `ancestor` axis contains the ancestors of the context node where the ancestors of a node are the parent and the parent's parent and so on. The `ancestor` axis always includes the root node unless the context node is the root node itself.
- The `ancestor-or-self` axis contains the context node and the ancestors of the context node. The `ancestor` axis always includes the root node.
- The `following-sibling` (`preceding-sibling`) axis contains all the following (preceding) siblings of the context node respectively. If the context node is an attribute node or namespace node, the `following-sibling` and `preceding-sibling` axes are empty.
- The `following` (`preceding`) axis contains all nodes in the same document as the context node that are after (before) the context node in document order, excluding any descendants, attribute nodes and namespace nodes.
- The `attribute` axis contains all the attributes of the context node. If the context node is not an element then the axis will be empty.
- The `namespace` axis contains the namespace nodes of the context node. If the context node is not an element then the axis will be empty.

For selecting element nodes, names of nodes or the wildcard character "`*`" can be used as *node tests*. A node test "`*`" is true for any node of the corresponding node type. For example, `child::*` will select all element children of the context node, and `attribute::*` will select all attributes of the context node. The `text()` node test is true for any text node. Finally, a node test `node()` is true for any node of any type.

We now continue with the *predicate* part of an XPath location step where many kinds of different expressions can be used. Predicates may consist of various functions operating on a set of nodes (e.g., `position()` function returns the position of the context node and `count(`*node-set*`)` returns the number of nodes in the respective node set). In addition, predicates may specify constraints on the textual data of the nodes (e.g., `article[title =` "*Distributed XML filtering*"`]` selects all `article` elements that have a child element `title` with the respective value. We call predicates of this type, *textual predicates*. Likewise, we may have predicates for the attributes of the nodes (e.g., `descendant::article[attribute::key =` "*conf/www/MKK08*"`]` selects all descendants of the context node that are `article` elements and whose `key` attribute has the value *"conf/www/MKK08"*). Predicates of this type are usually called *attribute predicates*. In the case of multiple adjacent predicates associated

with a location step, the predicates are applied from left to right, and the result of applying each predicate serves as the input for the next predicate.

2.1.2.3 Syntax

The syntax for a location step is the axis name and node test separated by a double colon, followed by zero or more expressions, each in square brackets. For example, consider query Q_1:`descendant::dblp/child::inproceedings`, which selects all the `inproceedings` elements that have a `dblp` parent element. Query Q_1 consists of two location steps. A most common way to write XPath queries is using an abbreviated form of syntax. The most important abbreviations, also used throughout this thesis, are the following:

- Location step `child::`*childname* can be abbreviated as *childname*. Actually, `child` is the default axis. For example, a location path `dblp/phdthesis` is short for `child::dblp/child::phdthesis`.

- Pattern `/descendant-or-self::node()/` can be abbreviated as `//`. For example, `//a` is short for `/descendant-or-self::node()/child::a` and selects any `a` element in the document.

- There is also an abbreviation for attributes where location step attribute::*childname* can be abbreviated to @*childname*. For example, a location path `inproceedings[@key="conf/www/MiliarakiKK08"]` is short for `child::dblp[attribute::type="conf/www/MiliarakiKK08"]` and so selects `dblp` children with a `key` attribute with value equal to `"conf/www/MiliarakiKK08"`.

- A location step of `.` is short for `self::node()`. This becomes particularly useful in conjunction with `//`. For example, the location path `.//article` is short for `self::node()/descendant-or-self::node()/child::article` and so will select all `article` descendant elements of the context node.

- Similarly, a location step of `..` is short for `parent::node()`. For example, `../title` is short for `parent::node()/child::title` and so will select the `title` children of the parent of the context node.

In this section, we described the main characteristics of XPath query language focusing on its syntax and providing many examples. For the methods described throughout this thesis, we consider a subset of XPath query language which we formally define later in Chapter 3.

2.2 Finite automata

Finite automata are a useful computational model for many important kinds of hardware and software [43]. In this section, we define finite automata formally, distinguishing between deterministic and nondeterministic ones and also present some example applications. Before proceeding to a formal definition, let us first describe informally the notion of finite automata.

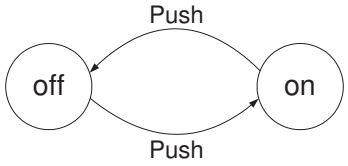


Figure 2.4: A finite automaton for a simple on/off switch

Each system can be viewed as being at all times in one of a finite number of *states*. For example, an on/off switch of a device can be viewed as one of the simplest nontrivial finite automaton [43]. Such a device remembers whether it is in the "on" state or the "off" state and the effect of pushing the button of the device is different depending on the current state of the switch. The corresponding finite automaton modeling this switch is depicted in Figure 2.4. States are represented using circles and the two states of this automaton are labeled "on" and "off". An arc represents an external influence on the system, in our case the user who pushes the button. Depending on these actions, the state the system is in changes.

Finite automata are classified into *deterministic* and *nondeterministic* [43]. A deterministic finite automaton (DFA) is in a single state after reading any sequence of inputs. In other words, deterministic refers to the fact that there is a single state to which the automaton can transition from its current state. In contrast, nondeterministic finite automata (NFA) can be in several states at once. Usually, when one refers to a finite automaton without specifying its class, it refers to a deterministic finite automaton. The automaton depicted in Figure 2.4 is a deterministic one.

2.2.1 Deterministic finite automata

We now proceed with a formal definition of finite automata starting with the deterministic ones. A *deterministic finite automaton* (DFA) [43] consists of:

1. A finite set of *states*, often denoted Q.
2. A finite set of input symbols, often denoted Σ, called the *alphabet*.
3. A *transition function*, commonly denoted as δ, which takes as arguments a state and an input symbol and returns a state. In our previous graph representation of automata,

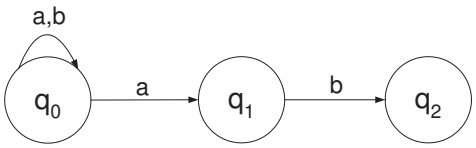


Figure 2.5: An NFA accepting all strings that end in ab

δ was represented by arcs between states and their respective labels. So, if there is an arc from a state q to a state p labeled with the input symbol a, it means that if the automaton is in state q when it reads an input symbol a, it moves to state p. We use the transition function to indicate this as $\delta(q, a) = p$.

4. A *start state*, one of the states in Q.

5. A set of *final* or *accepting states* F where $F \subseteq Q$.

A common notation for a DFA is using a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where $q_0 \in Q$ is the *start state*. The *language* of the DFA is the set of all strings that the DFA *accepts* or *recognizes*. To define the language of the DFA, we use an extended transition function that describes what happens after reading a sequence of inputs. If δ is the transition function, then the extended one constructed from δ is called $\hat{\delta}$. The extended transition function takes as input a state q and a string w and returns a state p, which is the state reached by the automaton when starting from state q and processing the sequence of inputs w. So, formally, the *language* $L(A)$ of a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is $L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ is in } F\}$. In other words, the language of A is the set of strings w that take the start state q_0 to one of the accepting states.

2.2.2 Nondeterministic finite automata

We continue now with the class of nondeterministic finite automata which have the power of being at several states at once. NFAs are considered easier to design and each NFA can be converted to an equivalent DFA. However, the latter may, although rarely, have exponentially more states than the NFA. In the worst case, the smallest DFA can have 2^n states while the smallest NFA for the same language has only n states [43]. Like the DFA, an NFA has a finite set of states, a finite set of input symbols, a single start state and a set of accepting states. The difference between an NFA and a DFA is in the transition function δ. In the case of the NFA, δ is a function that takes a state and input symbol as arguments and returns a set of zero, one, or more states instead of a single state in the case of the DFA. Recall that function δ for a DFA returns exactly one state. We show an example NFA in Figure 2.5.

Analogous to the formal definition of a DFA, we formally define an NFA as follows. A *nondeterministic finite automaton* (NFA) [43] consists of:

1. A finite set of *states*, often denoted Q.

2. A finite set of input symbols, often denoted Σ, called the *alphabet*.

3. A *transition function*, denoted as δ, which takes as arguments a state in Q and a member of Σ and returns a subset of Q.

4. A *start state*, denoted as q_0 and a member of Q.

5. A set of *final* or *accepting states* F, where $F \subseteq Q$.

A common notation for an NFA is using a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$. So, the NFA of Figure 2.5 can be specified formally as $(\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$. We also describe an extension of finite automata by allowing transitions on the empty string ϵ. As a result, an NFA is allowed to make an transition without receiving an input symbol. Typically, NFAs allowing ϵ-transitions (transitions without receiving an input symbol) are called ϵ-NFAs. The only difference between an NFA and an ϵ-NFA has to do with the transition function of the ϵ-NFA which takes as arguments a state in Q and a member of $\Sigma \cup \epsilon$. Throughout this thesis we will use the term NFA for actually referring to an ϵ-NFA.

Likewise to the language of a DFA, we define the *language* $L(A)$ of an NFA $A = (Q, \Sigma, \delta, q_0, F)$ is $L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq 0\}$. $L(A)$ is the set of strings w in $\Sigma \cup \{\epsilon\}$ such that $\hat{\delta}(q_0, w)$ contains at least one accepting state, where $\hat{\delta}$ is the extended transition function constructed from δ. Function $\hat{\delta}$ takes a state q and a string of input symbols w, and returns the set of states that the NFA is in, if it starts in state q and processes the string w. We also define the concept of ϵ-*closure* of a state q [43]. Let q be a state. The ϵ-closure of q, denoted by $\epsilon Close(q)$, is the set of all states that can be reached from q along any path whose arcs are all labeled ϵ.

The main idea behind how an NFA is executed is described by Sipser [91] as follows. Suppose that we are running an NFA and arrive at a state with more than one ways to proceed. After reading the corresponding symbol, the machine splits into multiple copies of itself and follows all the possibilities in parallel. Each copy of the machine continues as before and in case more choices appear, the NFA splits again. Nondeterminism can be viewed as a kind of parallel computation where several processes can be running concurrently. This fundamental observation motivated us in studying distributed NFA execution in this thesis.

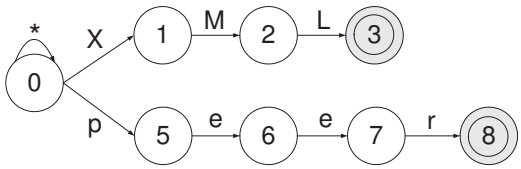


Figure 2.6: An NFA that searches for the words `XML` and `peer`

2.2.3 Applications

Finite automata are actually excellent models for several real problems. So, we conclude our discussion about finite automata by presenting some example applications. In our first example, inspired from the area of Web search, we demonstrate how we can perform text search with automata. In our second example application, we mention some indicative ways automata have been used for processing XML.

Text search The following example from the Web is described by Hopcroft et al. [43]. A common process found in the popular example of Web search engines is the following. Given a set of words, find all documents that contain one or all of these words. Search engines employ *inverted indexes*, i.e., indexes where for each word appearing on the Web (there are more than 100,000,000 different words), a list of all the Web places where that word occurs is stored. The most common of these lists are kept in the main memory of machines allowing fast search. Such indexes do not make use of finite automata but a lot of time is required to build them. However, similar applications that have certain characteristics and make use of such a process can employ automata-based techniques. One example is when the repository where the search is conducted rapidly changes, e.g., daily searching news articles for interesting topics.

Let us now describe how we can design an NFA for the above task. Suppose we are given a set of words, referred to as *keywords*, and we want to find occurrences of any of these words. For such application, it is useful to design a nondeterministic finite automaton for identifying when we have seen one of the keywords by entering an accepting state. The idea is that the text of a document is fed, one character at a time to the NFA to recognize occurrences of the keywords in this document. A simple form of an NFA that is able to recognize a set of keywords has the following characteristics [43]:

1. It has a start state with a transition to itself for every possible input symbol. Intuitively the idea is that the start state represents a guess that we have not yet seen one of the keywords, even if we have already seen some letters of one of these words.

2. For each keyword $a_1 a_2 \dots a_k$, the NFA includes k states, $q_1, q_2, \dots, q_k$. We add a transition from the start state to q_1 labeled with character a_1, a transition from q_1 to q_2

labeled with character a_2, and so on. The last state q_k for each keyword is an accepting state and indicates that we have found the corresponding keyword.

Example 2.2.1 (Text search with NFAs)**.** *Suppose we want to design an NFA to recognize occurrences of the words* `XML` *and* `peer`*. The NFA designed with the above characteristics is shown in Figure 2.6. State* 0 *is the start state and character* Σ *is used to represent the set of all allowed characters. States* 1, 2 *and* 3 *are used to recognize word* `XML`*, while states* 5, 6, 7 *and* 8 *recognize* `peer`*. Accepting states* 3 *and* 8 *(denoted by two concentric circles) indicate that the corresponding keywords have been found. It is interesting that a well-known text-processing program that uses a similar approach include advanced forms of the UNIX* `grep` *command (*`egrep` *and* `fgrep`*).*

XML processing As extensively surveyed by Schwentick [89], automata models are considered very useful for XML and there is a plethora of works that make use of automata in several ways. For example, Segoufin and Vianu [90] study the problem of validating streaming XML documents against a DTD using a finite automaton. For this purpose they employ a finite automaton and use constant memory to achieve this.

In this thesis, we use an NFA for performing XML filtering in a distributed environment. More details about our representation and how we implement such a method are described later in Chapters 3 and 4.

2.3 Peer-to-peer networks

Back in 1999, Napster[4] pioneered the idea of peer-to-peer (P2P) file sharing systems using the killer application of digital music sharing. Even though Napster supported a centralized search functionality, it was the first system to acknowledge that requests for popular content can be handled by many peers instead of being sent to a central server. Following this paradigm, other systems like Gnutella [2] which distributed both search and download capabilities, Kazaa[5] and Morpheus[6] indicate the immediate success of this approach. Nowadays, ideas from P2P computing have been applied to many popular distributed applications beyond traditional data sharing such as Grid computation (e.g., SETI@Home[7], IP telephony (e.g., Skype[8]) and content distribution.

[4] `http://www.napster.com`
[5] `http://www.kazaa.com`
[6] `http://www.musiccity.com`
[7] `http://www.setiathome.ssl.berkeley.edu`
[8] `http://www.skype.com`

In this section, we describe the technology of P2P networks. P2P networks are distributed systems consisting of a very large number of computing nodes that cooperate for sharing data without any centralized control. They go beyond the typical client-server systems since each peer can play both the role of a client and a server. Some of the most desirable features of peer-to-peer networks include robustness, efficient search of data items, redundant storage, hierarchical naming, anonymity, massive scalability and fault tolerance.

P2P overlay networks are typically classified into two main classes, *structured* and *unstructured* P2P networks. In the following, we first briefly describe unstructured P2P networks and then emphasize on structured ones and specifically on distributed hash tables (DHTs). We briefly survey some well-known DHTs including Pastry [88] which is one of the earlier influential DHTs and forms the basis of this research. This study hardly represents a comprehensive survey of the area and the interested reader can see the study of Lua et al. [60] and other more detailed surveys [10, 101] for this purpose.

2.3.1 Unstructured P2P systems

This class of P2P networks corresponds to the case where the nodes are organized without any restrictions on the topology or in a hierarchical manner. In the latter case super-peer networks introduce a hierarchy of peers classifying them into *super-peers* and *clients*. All super-peers are equal and have the same responsibilities. Each super-peer serves a fraction of the clients and typically holds an index for this purpose so that the clients can learn about resources by querying them. Clients are also equal to each other and actually keep the resources. In these overlays, search is performed using flooding, random walks or techniques like expanding ring search.

Napster utilized an unstructured P2P network requiring at the same time a centralized server for providing search capability (i.e., the directory server). Also, Gnutella [2] was the first system to make use of a fully distributed unstructured P2P network. Gnutella used flooding to send queries across the network within a defined limited scope. However, such flooding techniques may be effective for locating highly replicated data items and highly resilient to churn but they perform poorly for locating unpopular items and can lead to a high consumption of network bandwidth.

2.3.2 Structured P2P systems

Structured networks have a network topology that is tightly controlled and the content of the peers is not placed at random peers but at specified locations for more efficient performance. Such structured networks use the distributed hash table (DHT) paradigm where each data

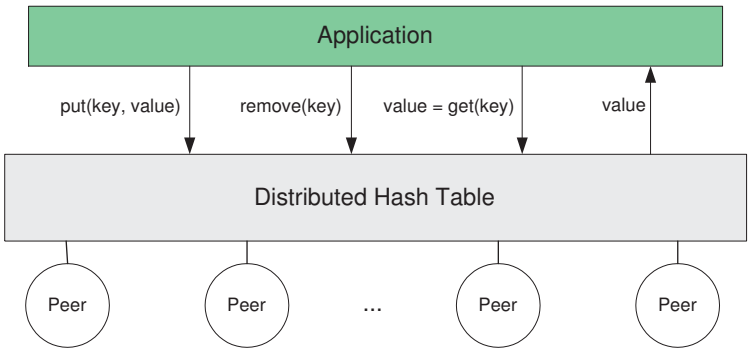


Figure 2.7: Application interface for distributed hash tables

item is placed deterministically at a peer. The roots of DHTs can be found in work conducted some years ago in the area of distributed data structures and databases. Litwin et al. [58] coined the term scalable distributed data structures (SDDS) for introducing a class of data structures used in distributed environments. They introduced LH*, a generalization of linear hashing (LH) for distributed RAM and disk files. LH [58] does not require a central directory and allows a file to extend to any number of sites. Although DHTs and SDDS offer a similar abstraction, their application area differs significantly and each system makes different assumptions for the reliability and dynamicity of the underlying network system. DHTs were designed and deployed for highly dynamic environments where peer failures are common. This assumption led to designs in which each peer requires to know only a limited number of other peers in the system (e.g., $O(\log N)$). Thus, any change in the system such as a peer joining or leaving affects only a relatively small number of other peers.

DHTs are structured P2P systems which aim to solve the following data location problem in a network of peers: *Suppose x is a data item stored at a distributed dynamic network of peers, find data item x.* The core idea in all different DHTs is to solve this search problem by offering a kind of distributed hash table functionality where data items are uniquely identified by keys and DHT peers cooperate to store these keys. Although there have been many proposals of DHTs like Chord [93, 94], Pastry [88] and CAN [85] which differ in their technical details, they all provide a very simple interface consisting of two operations:

- `put(`k`, `v`)`, this operation inserts a data item identified by key k and value v in the DHT.

- `get(`k`)`, this operation retrieves the data items associated with key k from the DHT node responsible for key `k`.

Also a `lookup(`k`)` operation returns a pointer to the DHT node responsible for key k. A generic interface demonstrating the operations supported by DHTs, also including a `remove`

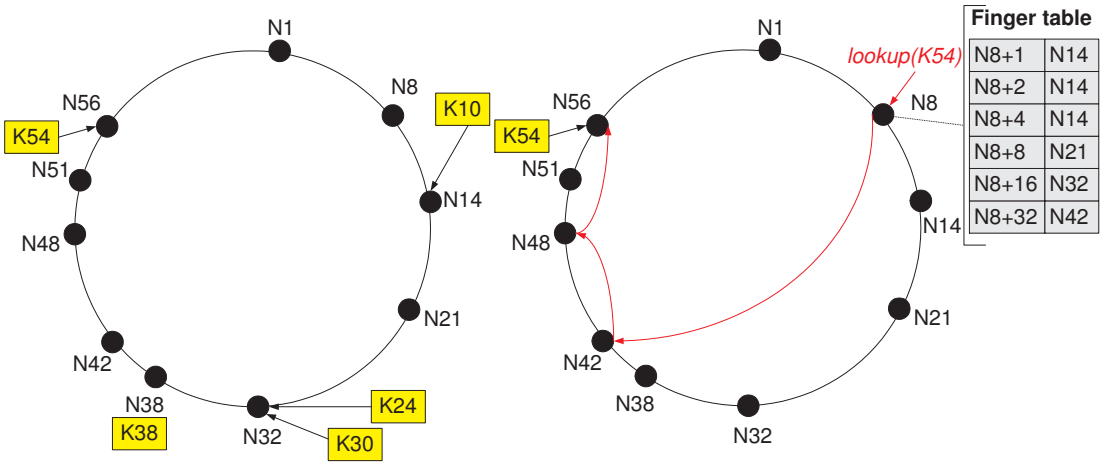


Figure 2.8: Chord ring with 10 nodes storing 5 keys (adapted from [93])

operation which deletes keys from the network, is shown in Figure 2.7 (adapted from the survey of Lua et al. [60]). A value can be any kind of data item including a document, a file or an address. Chord can easily implement this functionality by storing each key/value pair at the node to which that key maps. In the following, we describe in detail Chord and Pastry which correspond to two of the most popular DHT examples. Other DHTs include Tapestry [110], Content Addressable Network (CAN) [85], Kademlia [64], Viceroy [62], Bamboo [87] and Skip Graphs [9].

2.3.2.1 Chord

Chord DHT [93, 94], proposed by Stoica et al., uses a variant of consistent hashing [51] to assign keys to peers. Consistent hashing is designed to allow peers joining and leaving the network without significantly changing the mapping of keys. Specifically, only K/n keys need to be remapped on average, where K is the number of keys, and n is the number of peers. In contrast, traditional hash tables would require nearly all keys to be remapped in such a case. Using this decentralized scheme, each peer receives roughly the same number of keys and whenever peers join or leave the network there is a small relocation of keys. Chord improves even more the scalability of consistent hashing by avoiding the requirement that every node knows about every other node and for a system of N peers, each peer needs to maintain routing information for only $O(\log N)$ other peers.

In the consistent hashing scheme each node and data item is assigned an m-bit identifier using SHA-1 [71] as the base hash function, where m should be large enough so that the probability of keys hashing to the same identifier is negligible. The identifier of a peer is computed by hashing its IP address. Keys are assigned to data items and their identifiers are obtained by hashing these keys. In a file-sharing scenario, each file may use as key its

name but in general this is an application-specific decision and does not affect the scheme. Identifiers are ordered on an identifier circle modulo 2^m i.e., from 0 to $2^m - 1$. The identifier circle is also termed as the *Chord ring*. A key k is assigned to the first peer whose identifier is equal to or follows the identifier of k (also denoted as $H(k)$, where H is the SHA-1 hash function) in the identifier space. This node is called the *successor* node of key k and is denoted as *successor(k)*. If identifiers are represented as a circle of number from 0 to $2^m - 1$, then *successor(k)* is the first number appearing clockwise from k. We will also refer to a successor peer of a key as the *responsible* peer for this key.

If each peer knows only its successor, a query for locating the peer responsible for a key k can always be answered in $O(N)$ steps where N is the number of network peers. Chord improves on this bound, by having each peer maintaining a routing table, called the *finger table*, with at most m entries, where m is the number of bits in the identifier space. Each entry i in the finger table of peer n, points to the first node s on the identifier circle that succeeds identifier $H(n) + 2^{i-1}$. These nodes (i.e., $successor(H(n) + 2^{i-1})$ for $1 \leq i \leq m$) are called the *fingers* of node n. Since fingers point at repeatedly doubling distances away from n, they can speed-up search for locating the node responsible for a key k. If the finger tables have size $O(\log N)$, then finding a successor of a node n can be done in $O(\log N)$ steps with high probability [93]. In addition, when a peer n leaves Chord, all of its assigned keys are reassigned to n's successor. Therefore, a join or leave operation requires $O(\log^2 N)$ messages.

Figure 2.8 depicts a Chord ring where $m = 6$. This ring consists of 10 peers and 5 keys are stored. The successor of the identifier 10 is peer 14, so key is located there. In case a peer with identifier 26 joined the network, key with identifier 24 would be stored there instead of peer with identifier 32. We also illustrate how a lookup is performed. Peer 8 performs a lookup for key 54 and the path of the query by utilizing the entries of the finger table is depicted in the figure.

2.3.2.2 Pastry

Pastry [88], along with Tapestry [110], employ routing based on address prefixes. Such an approach bears some similarity with the distributed data structure proposed by Plaxton et al. [81], known as Plaxton mesh. In Plaxton mesh, a *hypercube routing* algorithm is used to efficiently locate a resource by utilizing routing tables of small size stored in each node in the network. Pastry [88] uses a version of this algorithm as the core of its routing mechanism, modified appropriately for a dynamic environment. The Chord protocol is closely related to both Pastry and Tapestry, but while Pastry routes towards peers that share successively longer address prefixes with the destination, Chord forwards messages based on numerical

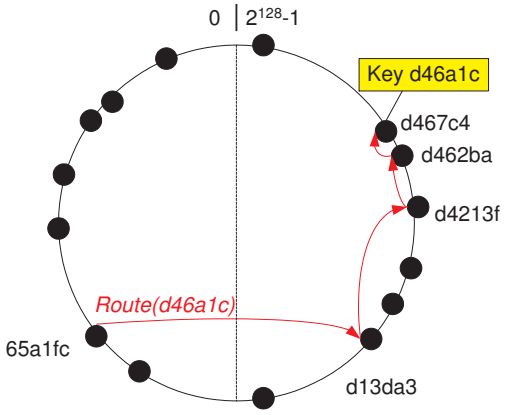


Figure 2.9: Pastry circular nodeId space: Routing a message

difference with the destination address.

Each peer in Pastry is assigned a 128-bit node identifier, referred to as *NodeId*. The NodeId is used to indicate the position of the node in a circular NodeId space ranging from 0 to $2^{128}-1$. The NodeId is assigned randomly when the node joins the network. Likewise to other DHTs, a cryptographic hash of the IP address of the node can be used for generating the identifier. The random way of assigning a NodeId to peers results with high probability in nodes with adjacent node identifiers being diverse in characteristics like location and ownership.

Assuming a network consisting of N nodes, Pastry routes to the numerically closest peer to a given key in less than $\log_B N$ steps under normal operation. Configuration parameter $B = 2^b$ has a typical value where $b = 4$. Each peer maintains as routing states, a routing table, a neighborhood set, and a leaf set. The routing table includes $logBN$ rows, each one including $B-1$ entries. The IP address of peers is kept with each entry.

2.4 Related research

In this section, we provide a detailed survey of previous research relevant to the topic of this thesis. We begin our discussion with centralized XML filtering systems and continue with distributed ones. We give an extensive survey of distributed approaches, discuss their main properties and also refer to whether they deal with load balancing in their settings. In addition, we comment on whether they explicitly consider predicate evaluation and discuss the relevant methods. Next, we give a small overview of distributed systems offering XML query processing functionality (i.e., one-time query processing). Finally, since our system FoXtrot, designed and developed in the context of this thesis, is built on top of a DHT, we also give a comprehensive survey of systems supporting different data models in a DHT

environment.

2.4.1 XML-based publish/subscribe systems

As we discussed earlier, in XML filtering systems deployed on the Internet, we expect that the amount of queries to be evaluated is very large. Consequently, it is considered prohibitively expensive to use a brute force approach and evaluate the queries one at a time. Based on this insight, an important optimization for XML filtering systems has been to build an index over the queries and actually evaluate a smaller set of queries against the XML documents. There is a significant amount of work that propose such indexes in centralized environments and also others that employ them in a distributed setting. With respect to predicate evaluation, while most of the systems, both centralized and distributed ones, support both structural and value matching, they focus either on structural matching or on the evaluation of value-based predicates. We also focus on the latter systems and describe their techniques. In the following we give a detailed survey of these works, point out some of their weaknesses which have motivated the research described in this thesis.

2.4.1.1 Centralized approaches

Many approaches have been proposed in the past for XML filtering in a centralized setting. A popular approach in many of the earlier works has been to adopt some form of finite state machine (FSM) to represent each path expression [7, 47]. This approach is based on the observation that a path expression can be transformed into a regular expression and consequently there exists an FSM that accepts the language described by the path expression [43]. Both XFilter [7] and Tukwila [47] create one FSM for each path expression by mapping its location steps to machine states. XML documents that arrive are parsed using an event-based parser (e.g., SAX parser [65]) and the events are used to drive the execution of query FSMs. A path expression is said to match a document if during parsing, the accepting state for that path is reached. For this purpose XFilter builds a dynamic index over the states of query FSMs which changes its content as parsing events drive the execution of the FSMs. However, by creating a single FSM per query, such an approach fails to take into account potential commonalities among the queries and it is subject to scalability problems since a large amount of redundant work may be performed.

Diao et al. [30] overcome XFilter weaknesses by constructing a single nondeterministic finite automaton (NFA) from the set of XPath queries. The NFA is used as a matching engine that scans incoming XML documents and discovers matching queries. YFilter is considered a state-of-the-art filtering engine exhibiting high performance for large sets of queries due to exploiting common path prefixes. With respect to predicate evaluation, the

authors propose two methods in YFilter, namely *Inline* and *Selection-Postponed*. Method Inline processes predicates as soon as the relevant state is reached during NFA execution while Selection-Postponed delays predicate evaluation until after the whole structure of a query is matched.

Zhang et al. [109] investigate an optimization of YFilter algorithm, called YFilter*. The authors employ clustering and aggregation methods aiming to decrease the number of matchings performed and improve the performance of the system. To accomplish this the authors design a new metric for measuring the similarity among query patterns.

Gupta and Suciu [41] identify the need for eliminating redundant work not only in the structural part of the queries but also in the predicate evaluation part. The authors propose to construct a single deterministic pushdown automaton, called XPush machine, and perform structural matching along with predicate evaluation directly with this automaton. They avoid the theoretical exponential state blow-up by computing the XPush machine lazily also consider some heuristic-based optimizations to reduce the total number of states. Their approach scales for both a large number of XPath queries and a large number of predicates per query.

Chan et al. [20] utilize a trie-based data structure called XTrie aiming to reduce unnecessary index probes and avoid redundant matchings. XTrie indexes tree-structured XPath queries by first decomposing them into a minimal number of substrings, each being a nonempty sequence of elements, which only contain parent-child relationships. These substrings are then organized using a trie structure and an auxiliary table which contains an entry for each substring of each indexed XPath query. In each row of the table, a set of values describes the positional and structural constraints of the associated substring.

Another filtering system called FiST is proposed by Kwon et al. [54]. The authors perform holistic matching of tree-structured XPath patterns (also called *twigs*) against incoming XML documents. Instead of splitting the queries into linear paths and then performing the matching, Kwon et al. [54] transform twigs into Prüfer sequences and then match each twig as a whole. The authors demonstrate that FiST outperforms YFilter in terms of scalability under various situations. In a more recent work, Kwon et al. [55] study an extension of FiST and design pFiST (predicate enabled FiST) to also support value-based predicates in the twig queries. The method proposed evaluates queries in a bottom-up fashion, checking the value-based predicates before performing structural matching. The authors evaluate their method comparing to YFilter for the case of equality predicates and report that pFiST exhibits a better performance.

A different approach is followed by Tian et al. [98] who design and implement an XML publish/subscribe system using a relational database. The authors deal with both structural and value matching by breaking queries into two parts, namely the predicate matching part

and the XML tree structure. Other systems offering centralized XML filtering include work of Index-Filter [16] where the authors build indexes not only for the queries but also for the incoming XML documents, XSQ system [74, 75], XSM machine [61], BoXFilter [70] and more recently AFilter [17] which exploits both prefix and suffix commonalities.

Since centralized solutions typically suffer from disadvantages including lack of scalability, zero fault tolerance and creation of bottlenecks, in the following section we continue our discussion with distributed approaches.

2.4.1.2 Distributed approaches

The majority of distributed approaches [92, 24, 33, 29, 38, 102, 21] assume a content-based overlay network where routers are responsible for forwarding XML data towards interested subscribers. Content-based routers, also called brokers, route XML data based on their content and are organized in mesh or tree-based configurations. We will describe in detail the most distinctive works emphasizing on the specific properties of network setting considered, the methods employed by the network brokers for forwarding XML data to interested subscribers, methods for dealing with predicate evaluation and other interesting properties like load distribution among the network brokers. With respect to predicate evaluation, we also include works that focus explicitly on the evaluation of value-based predicates. We also survey the latter systems emphasizing on these techniques. We also include works that focus on optimizing the functionality of each single XML broker and can be considered complementary to most of the other approaches.

In the work of Felber et al. [33], the authors first propose two simple strategies for parallelizing the filtering task, performed using the XTrie algorithm [20], among the available XML routers. In the first strategy, called data-sharing, each router keeps the whole set of queries and a load balancer dispatches each XML document that arrives to one of the routers. In the second strategy, called filter-sharing, routers share the queries equally and incoming XML documents are filtered by all the routers. Intuitively, the time for indexing queries in the former strategy is proportional to the number of routers in the network, while filtering time decreases as more routers become available. In addition, filtering in the filter-sharing strategy requires a broadcast operation to all routers. Whereas these two strategies are proposed in the context of XTrie, similar approaches could be supported by any kind of filtering engine. Apart from these strategies, referred together as Parallel XTrie, the authors also propose to organize the XML routers in a hierarchical structure and deal with the challenge of partitioning queries among the network brokers using the XTrie structure. In the latter approach, the authors identify that certain routers may suffer heavy access load but do not offer solutions to this problem. An overview of the different strategies is illustrated

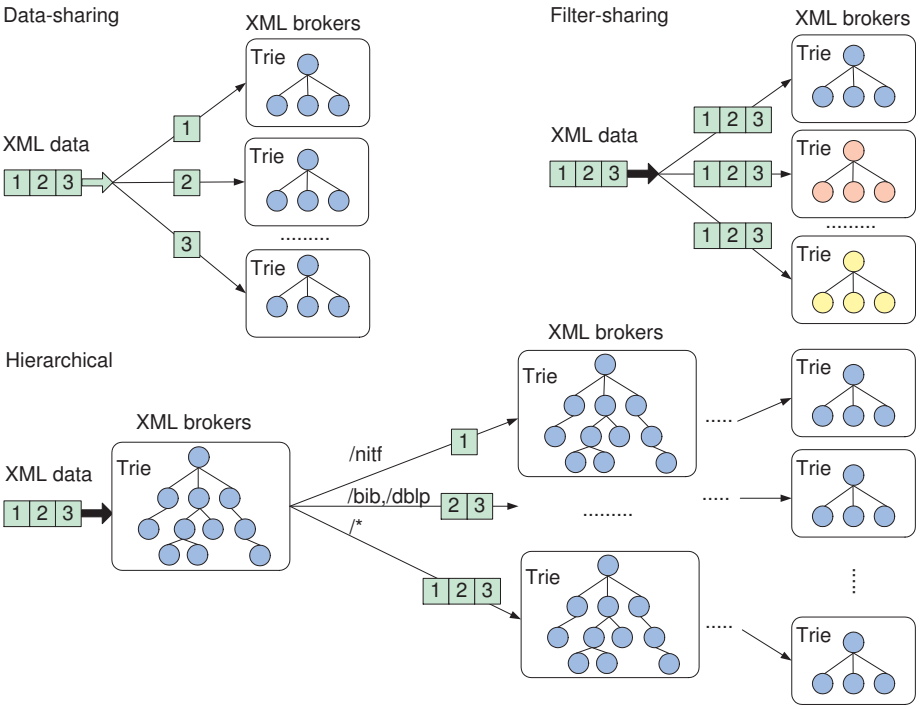


Figure 2.10: Parallel filtering strategies with XTrie [33]

in Figure 2.10.

In a more recent work by the authors of XNet [22], the authors describe another XML content-based network called XNet, where XML filtering is also performed using the XTrie algorithm [20]. A global spanning tree is used to implement a broadcast layer for publishers to communicate with all XML routers forming the inner network. XNet focuses on aggregation techniques for minimizing the size of the routing tables kept by the routers and employs fault tolerance methods to recover from router failures. The authors evaluate XNet using an overlay consisting of 22 nodes from the PlanetLab network and experiment with a subscription set containing at most 10^5 queries. They report on an indexing throughput of almost 19 single-element queries per second for each consumer node. While they report on the routing load of the individual nodes in terms of the size of their routing tables, the authors do not deal with any other kind of load with respect to load balancing.

We continue our discussion with the ONYX system, proposed by Diao et al. [29], which assumes a similar topology to XNet, where each broker uses a broadcast tree for reaching all other brokers in the network. ONYX consists of an overlay network of nodes. Most of the nodes serve as information brokers and handle XML messages and queries. A centralized component, called the registration service, is used to assign a priori XML data sources and queries to brokers using criteria like topological distances between the source and broker, available bandwidth, query content and the location of the subscriber. As a result, the

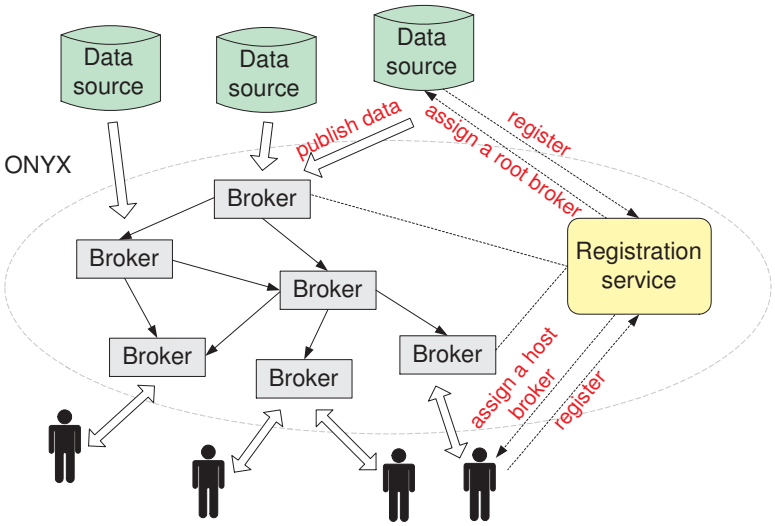


Figure 2.11: ONYX architecture [29]

registration service can suffer a lot of load. However, the authors do not address this issue or deal explicitly with the distribution of load among the network routers in general. An overview of the ONYX architecture is illustrated in Figure 2.11. The authors state that instead of a single node, the registration service may be offered by a small number of nodes which collaborate for this purpose. However, it still remains a single point of failure and a potential bottleneck for system ONYX even though some fault-tolerance can be achieved by having backup nodes. Diao et al. [29] consider this out of the scope of their research and do not study it any further. In ONYX, each message broker participating in the network runs a YFilter instance [30] for filtering and transforming XML messages. In addition, each broker contains a routing component to efficiently forward messages to the downstream brokers that are interested in the messages.

Another system called SONNET [111], is closely related to our work since it uses a DHT to build an XML dissemination system. In contrast to ONYX and XNet that forward XML data using queries as entries in the routing tables of the brokers, the authors of [111] construct a summarization of the queries using path digests. So, during the routing process, instead of performing XML filtering, checking is performed using cheaper bit-based operations. As a result, filtering is approximate and false positives are introduced. The authors use load-shedding techniques to balance the number of packets forwarded by each peer. However, they do not consider the size of each packet or the processing load suffered by each peer receiving a packet. They evaluate their system using a simulated peer-to-peer network.

The system XTreeNet [34] combines the publish/subscribe and the query/ response models within a network of XML routers that connects XML data producers and consumers. The

authors introduce the concept of content descriptors and create a different distribution tree for each of these descriptors. As a result, XML data are not matched repeatedly at internal brokers. Content descriptors can be elements of an ontological topic hierarchy or XML data paths and are considered high-level descriptions for both subscription and publication information.

We continue with a work that deals explicitly with load balancing issues [102]. The authors assume that they have available a number of servers that share XPath queries. Filtering is performed using a lazily constructed DFA similar to the work of Gupta et al. [41]. The authors employ a technique for transferring XPath queries from overloaded to under-loaded servers using a centralized component called XPE control server.

Papaemmanouil and Cetintemel [72] describe another relevant system called SemCast for distributed content-based routing. SemCast works with either relational or XML data, and in the case of XML, queries are expressed using the XPath language. In contrast to ONYX and other systems that require content-based filtering performed at all brokers, SemCast splits incoming data streams a priori and sends them across multiple channels implemented as independent dissemination trees. The process of deciding which and how many channels are created is called *channelization*. This process can be periodically revised and is based on criteria like network statistics, stream statistics and profile characteristics. The authors consider as a key advantage for SemCast that content-based filtering takes place only at the source and destination brokers and do not focus on which filtering engine is used for this purpose.

Also, other works in the literature [69, 21, 38] focus on optimizing the functionality of each single XML broker and can be considered complementary to most of the former works. For example, Gong et al. [38] describe techniques that use Bloom filters for summarizing the queries included in the routing tables. However, such a technique does not specify and is independent of the distributed setting used for organizing the brokers.

Vagena et al. [103] deal with XML message aggregation and describe the VA-RoXSum structure. VA-RoXSum aggregates structural information of XML data in a compact way, while Bloom filters are used to encode values of root-to-leaf XML data paths. Similarly to YFilter, in each broker queries are indexed using an NFA and value predicates are augmented with the final states of the NFA using Bloom filters. Also, Gong et al. [38] use Bloom filters to summarize path queries and build routing tables for efficiently filtering XML data. Finally, Bloom filters have been used by Pitoura et al. [53] as summaries of XML data to efficiently route path queries in a P2P network.

2.4.2 XML query processing in P2P networks

To offer a complete review of related work in this area, we point out that there are various interesting papers on storing XML documents in P2P networks and executing XPath queries. Representative works include approach by Bonifati et al. [14], the catalog service developed by Galanis et al. [36], and also work of Pitoura et al. [53]. In the latter, the authors study content-based routing for XPath queries in a P2P network storing XML documents [53]. Peers are connected using an unstructured P2P network and clustered based on their contents. We do not present an in-depth discussion of these papers since their emphasis is not on filtering algorithms.

2.4.3 Tree structures in DHTs

We now describe approaches that distribute tree-like structures on top of DHTs and other publish/subscribe systems built using peer-to-peer networks. These works do not necessarily consider the XML data model but since in FoXtrot we distribute an automaton on top of a DHT, we consider these works closely related and we discuss them in detail.

We begin with psiX [84], a hierarchical distributed index for locating XML data in a DHT network. Each XML document and query is mapped into an algebraic signature and indexes, called *H-indexes*, are built for the document signatures. To answer a query, first the root node of each H-index is discovered using the query elements. This is considered a special node and its id is computed based on the relevant XML element name. XML data location continues with each peer following pointers to the other nodes that keep index entries. Note that apart from locating the root node, where a DHT lookup operation is used, a peer continues traversing an H-index by following a set of extra pointers kept locally. Apart from disk usage, the authors do not study how load is distributed as a result of their design and depend solely on the underlying overlay offered by Chord for providing load balancing of key-value pairs.

A similar approach to psiX, was recently proposed by [5] with system KadoP which supports XPath query processing on top a DHT. Their indexing scheme is a combination of an inverted index on XML tags and a set of hierarchical indexes for storing the positional representation of tag name instances. Each peer keeps a list of indexes using B+-trees for XML tag elements. The authors acknowledge that distribution of element tags can be very skewed and peers migrate their data in the case of popular terms.

Prior to the above works that refer to the XML data model, [6] designed PastryStrings, where DHT peers also keep additional pointers for traversing a forest of trees representing a set of queries. PastryStrings supports queries expressed using an attribute-value data model handling a rich set of operators for both numerical and string attributes. Actually, they

consider an alphabet β, and for each character of β, a tree structure is created (called *string tree*) having *words* mapped to its tree nodes. Additional routing tables are kept by peers for enabling prefix-based routing. The authors are concerned with load balancing since as expected a fraction of nodes, the tree nodes close to the root of each tree, may become bottlenecks. For this reason, they use common used strategies including replication of the trees, partitioning of the storage load for popular values and also apply domain relocation techniques. The latter technique is based on the fact that each attribute is expected to have values from a very small part of its domain.

Zhang et al. [108] propose a distributed tree scheme called Brushwood, designed on top of Skip Graph DHT [9], where peers are assigned a tree partition using a linearization of the tree. The authors target locality-sensitive applications like distributed file services. This work is concerned with load balancing and uses load shedding methods to achieve this. In particular, peer-wise gossiping is used to aggregate load information inside the distributed tree and then trigger load adjustment operations.

One of the earlier approaches that uses a trie structure for organizing data in a peer-to-peer system is P-Grid system proposed by Aberer et al. [4]. P-Grid uses a virtual distributed search tree similarly structured as DHTs for supporting both prefix and range queries. Queries are resolved using prefix matching, while the actual network topology has no hierarchy and each peer keeps a part of the overall trie. For reasons including fault-tolerance and load-balancing, multiple peers are responsible for each leaf node of the P-Grid trie.

A different approach is proposed by Jagadish et al. [48], where the authors describe BATON (*BAlanced Tree Overlay Network*) for organizing peers in a distributed binary tree structure and supporting both exact match and range queries. In BATON, each peer stores tree nodes keeping links to its parent, children, adjacent nodes, and also some selected neighbors of the same level. For load balancing, again a load-shedding technique is used where overloaded nodes share or migrate their data. However such a technique is not sufficient when global imbalances occur. BATON* [49] improves on BATON by supporting multi-attribute queries using a multi-way tree structure. This improved design allows to achieve better load balancing by increasing the fanout of the tree leading to more leaf nodes. Also in this case load balancing occurs dynamically having peers partitioning or migrating their load when necessary. Other approaches include Prefix Hash Tree [82] that uses the lookup interface of a DHT to construct a trie-based structure for efficiently answering range queries.

We also mention that there are works that design publish/subscribe systems on top of DHTs supporting different data models. Such systems include SmartSeer system [50], Scribe [19], Corona [83] and the work of Tryfonopoulos et al. [100]. Corona [83] is a publish/subscribe system built on top of a DHT. In Corona, each information source is assigned to a random peer which monitors the source and disseminates updates to interested clients

who have subscribed to the specific source. The authors use an optimization method to decide which peers should monitor each channel using periodic polling aiming to optimize bandwidth utilization. In SmartSeer, the authors use a keyword-indexing method for allowing users to subscribe with queries containing conjunctions or disjunctions of terms over the CiteSeer database. Tryfonopoulos et al. [100] design an information filtering system supporting an attribute-value model in a DHT environment.

2.5 Summary

In this chapter, we described the XML data model and focused on XPath query language. Also, we gave an overview of finite automata and provided formal definitions of both deterministic and nondeterministic finite automata. Next, we presented the technology of peer-to-peer systems emphasizing on distributed hash tables. Finally, we provided a survey of work conducted in related areas focusing mainly on XML filtering systems. In the next chapter we give an overview of the distributed XML filtering system described and developed in this thesis, called FoXtrot. We describe the data model supported, its architecture and also present the API offered by our system.

Chapter 3

An XML filtering system

In this thesis, we design, develop and evaluate a fully distributed XML filtering system called *FoXtrot* (***F**iltering **O**f **X**ML data on top of s**TR**uctured **O**verlay ne**T**works*). FoXtrot exploits the technology of DHTs to offer XML filtering functionality on Internet scale. In this chapter, before describing our techniques in detail, we introduce the architecture of FoXtrot along with the data model it supports. We also present the FoXtrot API.

3.1 Data model

In this section, we describe our data model with respect to the XML documents and the subset of XPath we allow throughout this thesis for designing the methods we have developed in our system FoXtrot.

3.1.1 XML documents

We consider well-formed XML documents. XML documents may be valid according to a DTD or an XML schema but this is not required by our methods.

3.1.2 XPath queries

As described in detail in Section 2.1.2, each XPath expression consists of a sequence of *location steps*. Throughout this thesis, we consider location steps of the following form:

$$axis\ nodetest\ [predicate_1] \ldots [predicate_n]$$

where *axis* is a child (/) or a descendant (//) axis, *nodetest* is the name of the node or the wildcard character "*", and $predicate_i$ is a predicate in a list of one or more predicates used to refine the selection of the node. We allow either *attribute predicates* of the form [*attr op*

value] where *attr* is an attribute name, *value* is an attribute value and *op* is one of the basic logical comparison operators $\{=, >, >=, <, <=, <>\}$ or *textual predicates* of the form [*text*() *op value*] where *value* is a string value and *op* is a string operator as defined in XPath W3C recommendation [25].

A *linear path query* q is an expression of the form $l_1 l_2 \dots l_n$, where each l_i is a location step as defined above. In this thesis, queries are written using this subset of XPath, and we will refer to such queries as *path queries* or *XPath queries* interchangeably. We write path queries using the *abbreviated form of syntax* presented earlier. Queries containing branches can be managed by our algorithms by splitting them into a set of linear path queries.

Example 3.1.1 (Example path queries)**.** *We present here some example path queries for the case of a bibliographic database.*

Q_1: `/bibliography/phdthesis[@published>2007]`

which selects PhD theses published in year 2007.

Q_2: `/bibliography/*/author[text()="John Smith"]`

which selects any publication of author John Smith.

Q_3: `//*/[school="University of Athens"]`

which selects any type of publication written by authors from University of Athens.

Q_4: `/dblp/proceedings/author[position()=1]`

which selects all first authors of a publication.

3.2 An NFA-based XML filtering model

Automata and tree-based structures have been proven to be efficient ways for indexing path queries in XML filtering systems. As a result, we decided to use an NFA-based model, similar to the one use in YFilter [30], for indexing path queries in our approach and performing structural matching. Any path query can be transformed into a regular expression and consequently there exists an NFA that accepts the language described by this query [43]. Following YFilter [30], for a given set of path queries, we construct an NFA $A = (Q, \Sigma, \delta, q_0, F)$ where Σ contains element names and the *wildcard* (*) character, and each path query is associated with an accepting state $q \in F$. An example of this construction is depicted in Figure 3.1. The figure also shows how the different location steps in a query are represented using the corresponding NFA fragments. In FoXtrot, this NFA is distributed among the network peers which cooperate to perform XML filtering. Our techniques are described in detail in Chapter 4.

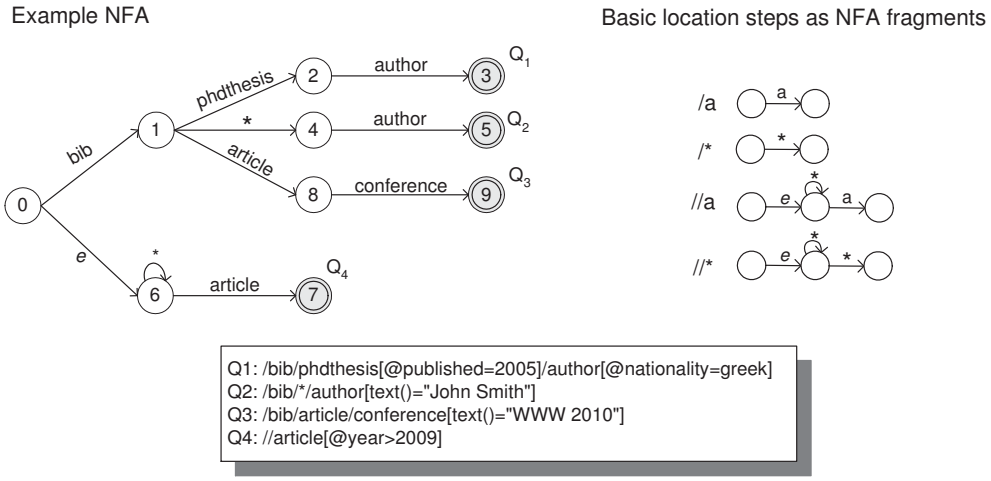


Figure 3.1: An example NFA constructed from a set of XPath queries

3.3 The FoXtrot architecture

Let us now describe the architecture of FoXtrot. A high-level view of the architecture is illustrated in Figure 3.2. FoXtrot is built on top of a P2P network where peers are organized using DHT-based protocols. Each peer participating in FoXtrot will be referred to as a *FoXtrot peer*. We can identify two parties involved, the *subscribers* and the *publishers*. In FoXtrot, subscribers submit path queries to the system asking to be notified whenever XML documents arrive that match these queries. Publishers publish their data in the form of XML documents. Each publisher/subscriber can communicate and submit its request at any FoXtrot peer participating in the network. Moreover, any FoXtrot peer can also play the role of a publisher or a subscriber.

We identify many application scenarios that can be supported by FoXtrot. For example, news monitoring where the providers of the news can be news agencies that provide XML feeds of their updates, common users or other parties which publish news. Another scenario we consider in this thesis is the publication monitoring scenario where a user wants to be notified about publications submitted to a bibliographic server.

3.4 FoXtrot API

We have developed an API in FoXtrot so that a user or an external party can communicate with any peer of FoXtrot, subscribe with queries or publish XML documents. The following functions are provided by the FoXtrot API:

- `subscribe(`*query*`)`, indexes a query q in the FoXtrot system. The query should be written in the subset of XPath defined earlier. For indexing the query, FoXtrot peers

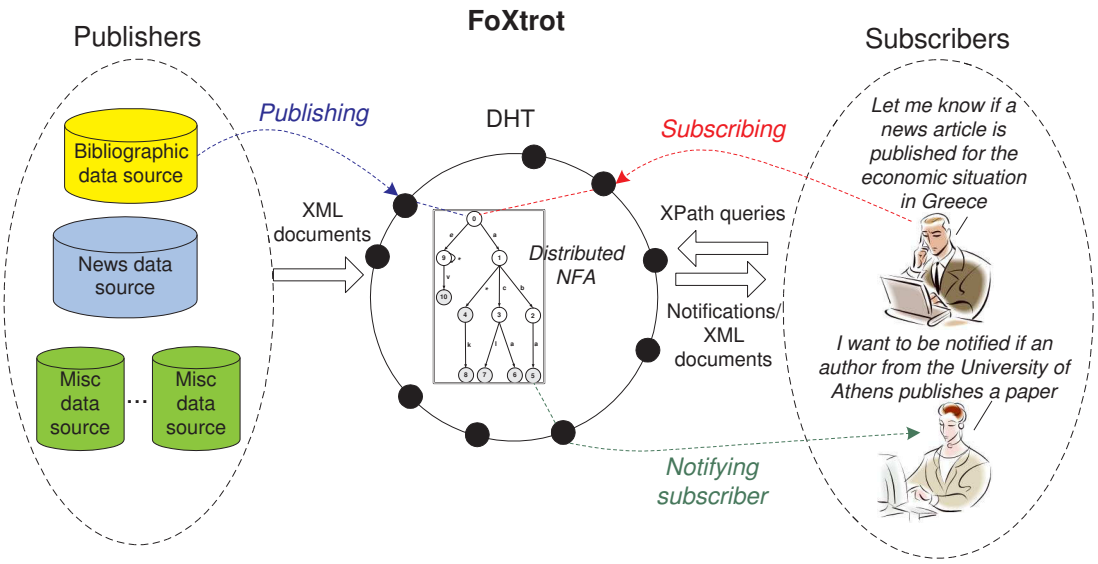


Figure 3.2: Architecture of FoXtrot

cooperate to traverse and update the distributed NFA. A pointer to the subscriber of q is associated with the query so that FoXtrot peers can communicate with her in case the query is satisfied.

- `publish(`*document*`)`, publishes an XML document d in FoXtrot. The XML document must be well-formed but not necessarily valid. FoXtrot peers cooperate to execute the distributed NFA using *recursive* method and if a match is discovered for a query, then a notification is dispatched to the interested subscriber. In the latter case, the peer who discovers the match, disseminates the XML document to the subscriber.

- `publish(`*document*`,`*method*`)`, publishes an XML document d in FoXtrot using the specified *method*. The available options that this call allows is the two supported methods, namely *iterative* and *recursive* method. These methods are described in detail in Chapter 4.

3.5 Summary

In this chapter we provided a high-level overview of FoXtrot, the distributed XML filtering system which is the subject of this thesis. We described the data and query model of FoXtrot, presented the architecture of the system and some example applications and also the supported API.

We continue with describing in detail the algorithms used by the FoXtrot peers for providing XML filtering functionality. First, we focus on how structural matching is performed

(Chapter 4) and then continue with value matching (Chapter 5).

Chapter 4

Structural matching

Automata and tree-based structures have been proven to be efficient ways for indexing path queries in XML filtering systems. For this reason, we decided to use an NFA-based model for query indexing and performing structural matching in our approach. The NFA corresponding to a set of path queries is essentially a tree-like structure that needs to be traversed both for indexing a query during NFA construction, and for finding matches against incoming XML data during NFA execution.

In FoXtrot, we distribute an NFA on top of Pastry and provide efficient ways of supporting these two basic operations performed by a filtering system, namely path query indexing and XML document filtering. Our main motivation for distributing the automaton derives from the nondeterministic nature of NFAs that allows them to be in several states at the same time, resulting in many different parallel executions. Apart from their nondeterministic nature, we also preferred to use an NFA instead of its equivalent DFA for reducing the number of states. Recall that in the worst case, the smallest DFA can have 2^n states while the smallest NFA for the same language has only n states [43].

In this chapter, we describe in detail how the NFA corresponding to a set of XPath queries is constructed, maintained and executed by the network peers for providing XML filtering functionality in the distributed environment of FoXtrot. In addition, we discuss load balancing methods for evenly distributing the load among the peers and also techniques for improving the fault tolerance of the system. We conclude this chapter by presenting an extensive experimental evaluation of FoXtrot.

4.1 Distributing the NFA

The distribution of the NFA among the network peers in FoXtrot is done at the level of NFA states. An example NFA for a small set of path queries is depicted in Figure 4.1. Since we

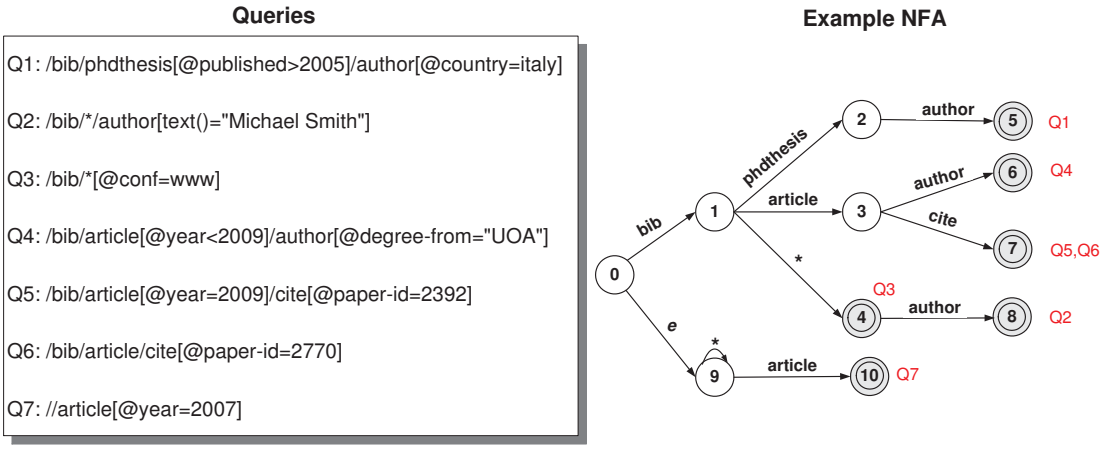


Figure 4.1: An example NFA constructed from a set of XPath queries

distribute the NFA on top of a Pastry network, we use the term *distributed NFA* to refer to it.

Each NFA state is assigned to a network peer as follows. Each state q_i along with every other state included in $\hat{\delta}(q_i, w)$, where w is a string of length l included in $\Sigma \cup \{\epsilon\}$, is assigned to a single peer in the network. Note that l is a parameter that determines how large part of the NFA is the responsibility of each peer. If $l = 0$, each state is indexed only once at a single peer, with the exception of states that are reached by an ϵ-transition, which are also stored at the peers responsible for the state which contains the ϵ-transition. Recall that the ϵ-transition represents a transition that can be followed without receiving an input symbol. Formally, if a peer is responsible for storing a state q, then it will also store the states included in the ϵ-closure of q, denoted by $\epsilon Close(q)$. For larger values of l, each state is stored at a single peer along with other states reachable from it by following a path of length l. Again, potential ϵ-transitions do not contribute to the specified length l. As a result, using parameter l results in storing each state at more than one peers. Therefore, peers store overlapping fragments of the NFA and parameter l characterizes the size of these fragments.

The use of parameter l creates a trade-off between the size of the NFA fragments that each peer holds and the degree of distribution of the NFA among the peers. The larger the value for parameter l is, each peer is aware of a larger part of the NFA and can perform a greater amount of work by itself decreasing the number of peers that are required to participate for completing a task (i.e., an indexing task or a filtering task). As we demonstrate later during the evaluation of FoXtrot, depending on the specific characteristics of the workload, tuning parameter l can lead to a significant improvement in performance.

We have explained so far how peers share the NFA states, having a single peer responsible

for each state. We now describe how we determine *which peer* will be responsible for a specific state. We uniquely identify each state with a key. The *responsible peer* for state with key k is the peer whose identifier is numerically closest to $Hash(k)$, where $Hash()$ is the DHT hash function. The key of an automaton state is formed by the concatenation of the transition labels included in the path leading to the state. For the NFA depicted in Figure 4.1, the key of state 4 is the string `start` + `bib` + $*$, the key of the start state is `start`, and state 9 has key `start` + `$`, since we choose to represent ϵ-transitions using character `$`. Operator + is used to denote the concatenation of strings.

Apart from parameter l, which adds redundancy to our system, we also employ replication techniques to achieve a balanced load. For example, even for larger values of l, given our distribution scheme as described above, the start state will be stored by a single peer creating a potential network bottleneck. For this reason, we study *load balancing methods* to distribute evenly the total load among the network peers. These methods are considered complementary to the methods we describe here for indexing XPath queries and filtering XML documents. For ease of the reader in understanding our basic methods, we first describe them without considering any load balancing method. Later, in Section 4.4 we discuss in detail how these methods are enhanced for achieving a better distribution of the load among the network peers.

Implementation To implement the NFA structure we use a hashing-based approach which has been shown by Watson et al. [105] to have low-time complexity for inserting states, inserting transitions, and following the transitions. For this purpose, each peer keeps the following data structures for holding each state and its associated information. The basic data structure is a hash table, denoted by $p.states$, which contains the states assigned to peer p indexed by their keys. For each state st included in $p.states$ we keep the transitions from st organized also in a hash table structure. The transition hash table for each state maps a string $label$ to an identifier st_{key}, where $label$ is the label of the outgoing transition (i.e., element name, $*$, or ϵ) and st_{key} is the identifier of the target state that the transition leads to. In the case of an ϵ-transition, we treat the child states differently since they include a self-loop labeled with $*$ (see state 9 as an example in Figure 4.1). For these states we do not index the self-loop in the transition hash table but keep it using a separate boolean parameter denoted as $st.selfChild$ for a state st. Finally, if st is an accepting state, we also keep the identifiers and the subscribers of the associated queries. Recall that a query matches a document if during the execution of the NFA, the accepting state for that query is reached. We denote the list containing the queries associated with an accepting state st, as $st.queries$.

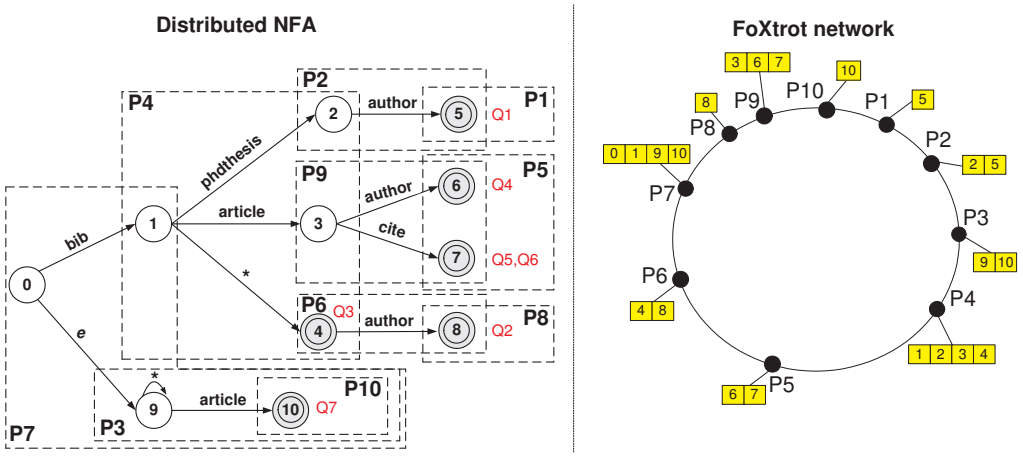


Figure 4.2: Distributing an NFA in FoXtrot ($l = 1$)

State key	0	1	2	3	4	5	6	7	8	9	10
Responsible peer	P7	P4	P2	P9	P6	P1	P5	P5	P8	P3	P10

Table 4.1: State assignments to peers

Example 4.1.1 (Distributing an NFA in FoXtrot)**.** *Figure 4.2 illustrates how we distribute an NFA on top of Pastry when* $l = 1$*. We consider a network of 10 peers and illustrate where each state is stored on the Pastry ring. Table 4.1 contains the peers responsible for each one of the states. We use unique integers instead of the actual state keys for readability purposes. The queries indexed in this example NFA are shown in Figure 4.1. Notice that state* 10 *is included in* $P7.states = [0, 1, 9, 10]$*, because the* ϵ*-transition does not contribute to the specified length* l*.*

4.2 Constructing a distributed NFA

We explained how the NFA corresponding to a given set of path queries is distributed among the DHT peers. To achieve this distribution in FoXtrot, we incrementally construct the automaton as XPath queries arrive in the system. In this section, we describe how this works in detail.

To help the reader understand this process, we first describe how the NFA is constructed without considering the fact that the states are distributed and stored at different peers. This process is identical to the construction process in the centralized environment of YFilter [30]. A location step in a query can be represented by an NFA fragment. The different location steps and the corresponding NFA fragments were shown in Figure 3.1. The NFA for a path query can be constructed by concatenating the NFA fragments of the location steps it consists of, and making the final state of the NFA the accepting state of the path query.

Inserting a new query into an existing NFA requires to *combine* the NFA of the query with the already existing one. So, to insert a new query represented by an NFA S to an existing NFA R, we start from the common start state shared by R and S and we traverse R until either we reach the accepting state of S or a state for which there is no transition that matches the corresponding transition of S. If the latter happens, a new transition is added to that state in R. Formally, if $L(R)$ is the language of the NFA already constructed by previously inserted queries, and $L(S)$ is the language of the NFA of the query being indexed, then the resulting NFA has language $L(R) \cup L(S)$. We show examples of how different NFA fragments are combined in Figure 4.3. We also depict how the previous example NFA of Figure 3.1 is updated after inserting query $Q5$.

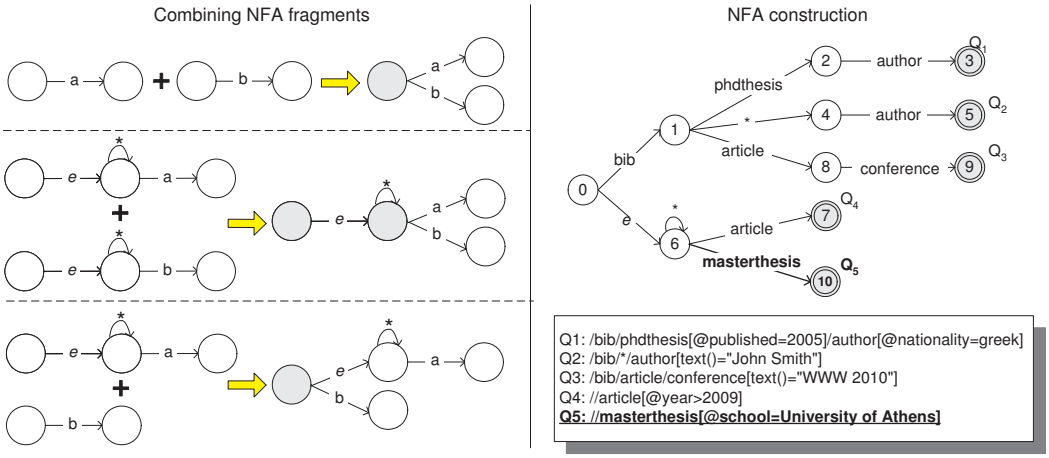


Figure 4.3: NFA construction

Let us now describe how we traverse the distributed NFA for inserting a query q. The main idea is that whenever we want to visit a particular NFA state st during indexing q, we first discover and contact the peer responsible for that state. If the state does not exist, the corresponding peer creates it. In case this state was previously created, then the peer checks whether there is the corresponding transition t associated with state st. If no such transition exists, t is added along with state st' where t leads to. Else, we follow transition t and indexing continues in the same way with the next state.

The exact steps followed are depicted in Algorithm 1. Algorithms are described using a notation, where p.`Proc()` means that peer p receives a message and initiates execution for procedure `Proc()`.

Suppose s is the subscriber peer for query q. Using Pastry, peer s sends a message IndexQueryMesg(q, d, st, sid) to peer r, where q is the query being indexed in the form of an NFA, d is the current depth of the query NFA reached, st is the state at this depth, and sid is the identifier of the subscriber peer. Initially $d = 0$, st is the start state, and

Algorithm 1: `IndexQuery()`: *Indexing a query*

```
1  procedure peer.IndexQuery(q, d, st, sid, l, firstCall)
2      if peer.states does not contain st then
3          peer.states.put(st.key, st) ;
4      else
5          st := peer.states.get(st.key);
6      if d == q.length then
7          add q to st.queries ;
8          peer.states.update(st.key, st) ;
9      else
10         t := transition label of q at depth d;
11         if no transition exists labeled t from st to st' then
12             add transition labeled t from st to st';
13             if t == $ then
14                 st' := st.getTransition(t);
15                 st'.selfChild := true;
16                 peer.states.put(st'.key, st');
17
18         else st' := st.getTransition(t);
19         if t == $ then
20             t' := transition label of q at depth d + 1;
21             if no transition exists labeled t' from st' to st'' then
22                 add transition labeled t' from st' to st'';
23
24         if firstCall is true then
25             nextPeer := Lookup(st'.key);
26             nextPeer.Route(IndexQueryMesg(q, d+1, st', sid));
27         else
28             if l > 0 then
29                 if t == $ then
30                     peer.IndexQuery (q, d+1, st', sid, l, false);
31                 else
32                     peer.IndexQuery (q, d+1, st', sid, l − 1, false);
33
34
35
36
37 end
```

r is the peer responsible for it. Starting from this peer, each peer r which receives an IndexQueryMesg message, executes locally the corresponding procedure `IndexQuery`(q, d, st, sid, l, $firstCall$), where l is the value of system parameter l and $firstCall$ is a boolean parameter initially true. If l is larger than 0, then r calls recursively this procedure to store locally the additional states as required by l. To distinguish between the first call of the procedure and a recursive one, we use parameter $firstCall$.

Let us now describe the details of the local procedure `IndexQuery` executed at each peer for a state st. At first, the peer checks whether state st is already stored locally. Recall that each NFA state is identified by a sequence of transition labels. If st is not stored locally, it creates it (lines 2-5). If st is the accepting state of q, q is inserted in the list $st.queries$ and execution ends (lines 6-8). At this point, the responsible peer can notify the subscriber peer that q is successfully indexed. Else, query indexing continues with the next state. Let t be the label of the transition from state st to state st'. Then, if there is no such transition from st, a new transition is added from state st to st' with label t (lines 11-12). If this transition is an ϵ-transition, then a self-loop transition is also added to state st' to represent a "`//`" step (lines 13-16). We also need to fix the local transition table of the next state st' (lines 19-22). Finally, peer creates and sends a new IndexQueryMesg(q, $d+1$, st', sid) message to the next responsible peer (i.e., the peer responsible for state st') increasing query depth by 1 and indexing continues in a similar (lines 24-26). If parameter l is larger than 0, the procedure is also called recursively l times by the peer to store additional states (lines 27-32).

Constructing the NFA as described above, requires sending as many IndexQueryMesg messages as the number of states in the NFA of query q. We clarify at this point that the number of messages that travel through the network during the construction of the NFA is independent of the value of parameter l. However, the value of l affects the time spent by each peer for locally processing the indexing request.

NFA updates As reported by Diao et al. [30], another great benefit of an NFA-based approach is the ease of maintaining it during insertion, updating or deletion of queries. First, for deleting a query q, we locate the accepting state of q, st and remove the identifier of the query from $st.queries$ list. In case this list becomes empty and the state has no children states in the NFA (i.e., contains no transitions), we can delete this state and also remove the corresponding transition from its parent. This deletion should also be propagated to its predecessors. Second, query updates can be handled by deleting the old queries and then inserting the updated ones.

The authors of YFilter [30] consider less expensive to adopt a lazy approach where a list of the deleted queries is maintained and checked before returning any results. The queries can then be deleted at a later point of time. However, it is not straightforward to adopt

such a lazy approach in FoXtrot where a large number of peers participates.

4.3 Executing a distributed NFA

In this section, we describe how the distributed NFA is executed for performing XML filtering, discovering matching queries and notifying the interested subscribers in FoXtrot. Again, to help the reader understand how this works, we first describe how the NFA would be executed if we ignore that the states are distributed. This process is similar to the execution process in the centralized environment of YFilter [30]. After explaining how we execute a centralized NFA, we describe two different methods for executing the distributed NFA, namely iterative and recursive method.

The NFA execution proceeds in an event-driven fashion. As the XML document is parsed, the produced events are fed, one event at a time, to the NFA to drive its transitions. Parsing is performed using a SAX (Simple API for XML) parser [65] that produces events of the following types: *StartOfElement, EndOfElement, StartOfDocument, EndOfDocument* and *Text*. During parsing, *StartOfElement* events trigger transitions of the NFA. The nesting of elements in an XML document requires that when an *EndOfElement* event is raised, the NFA execution should backtrack to the states it was in when the corresponding *StartOfElement* was raised. For achieving this, YFilter maintains a stack, called the *runtime stack*, during the execution of the NFA. Since many states can be active at the same time in an NFA, the stack is used for tracking multiple active paths. The states placed on the top of the stack will represent the *active states*, while the states found during each step of execution after following the transitions caused by the input event, will be called the *target states*.

Execution is initiated when a *StartOfDocument* event occurs and the start state of the NFA is pushed into the stack as the only active state. Then, each time a *StartOfElement* event occurs for an element e, all active states are checked for transitions labeled with e or *wildcard* ($*$). We also check for ϵ-transitions and in this case the target state is recursively checked one more time. All active states containing a *self-loop* are also added to the target states. The target states are pushed into the runtime stack and become the active states for the next execution step. If an accepting state is included in the active states, the NFA outputs the identifiers of all queries accepted at that state. If an *EndOfElement* event occurs, the top of the runtime stack is popped and backtracking takes place. Execution proceeds in this way until the document has been completely parsed or the stack becomes empty. Finally, it is important to note that, unlike traditional NFAs, whose goal is to find a single accepting state for each input, the NFA execution here must find all matching queries. So, even after we reach an accepting state during filtering an XML document, the execution

continues until the document has been completely parsed.

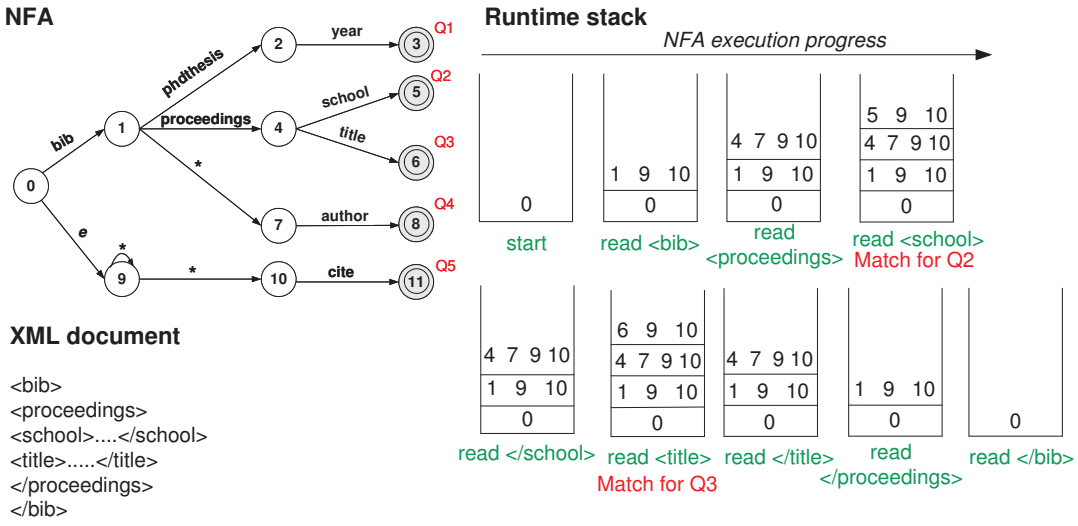


Figure 4.4: NFA execution (YFilter case)

Example 4.3.1 (NFA execution in YFilter)**.** *Let us consider the NFA depicted in Figure 4.4. The figure illustrates how the runtime stack is updated during NFA execution for filtering an XML document. Initially, only the start state,* 0*, is active. Parsing the start-tag for element* `<bib>` *causes transition to state* 1 *to be triggered and state* 1 *becomes active. States* 9 *and* 10 *also become active because of the* ϵ*-transition and the wildcard transition respectively. Later, when we parse the start-tag for element* `<school>`*, state* 5 *becomes active and therefore a match for query* $Q2$ *is detected since state* 5 *is an accepting state for this query. Whenever we read a close tag for an element, like element* `<proceedings>`*, the runtime stack is backtracked to the state immediately before reading the respective start-tag.*

Let us now proceed with the execution of a distributed NFA in FoXtrot. Likewise, we maintain a stack for backtracking during the execution. Also, for each active state we retrieve all target states reached after feeding the corresponding parsing event to the NFA. Given that the NFA states in our approach are distributed among the network peers, at each step of the execution, the relevant parsing event should be forwarded to the peers responsible for the active states. Having this in mind, we can identify two ways for executing the NFA: the first proceeds in an *iterative* way, while the other executes the NFA in a *recursive* fashion.

4.3.1 Iterative method

We begin our description with the *iterative* method. In this method, the publisher peer is responsible for parsing the XML document, maintaining the runtime stack, forwarding

the parsing events to the responsible peers and retrieving from them the target states to continue execution. As a result, the execution of the NFA proceeds in a similar way with the centralized case, except that the target states cannot be retrieved locally but are retrieved from other peers of the network. Furthermore in FoXtrot, we need to take into account parameter l which allows peers to keep larger fragments of the NFA. As a result, a peer responsible for an active state during execution can exploit the whole relevant NFA fragment (i.e., the NFA fragment beginning at that state) kept locally. In other words, each peer may perform several subsequent expansions instead of one. In the following we give a detailed description of this algorithm.

Description Algorithm 2 describes the actions required by the publisher peer during the iterative execution of the distributed NFA. The publisher peer p publishes a document by following the steps described in procedure `PublishIterative`(*doc*) where *doc* is the XML document being published. At first, p parses the XML document and stores the corresponding parsing events in a list (line 2). Then, it initializes a variable called *pathLength* using the value of parameter l. If l is equal to 0, then along with each active state the peer will send a single document element (from the parsing events) to the responsible peer. Else, if l is greater than 0, the publisher peer will send more than one elements with each active state, since the responsible peer will be able to perform more expansions (lines 4-7). At first p communicates with the peer responsible for the start state to retrieve it and then add it to the active states to initiate NFA execution (lines 8-11). Next, peer p begins reading the parsing events and inserts them in a list called *currentEvents* until either *pathLength* elements are inserted or an *EndElement* event is read (lines 12-20). Then, p sends a GetTargetStatesMesg message to each peer responsible for an active state. During the first iteration only the start state is included in active states. Each responsible peer proceeds with the expansion of the relevant states and returns the corresponding target states back to the publisher. Note that depending on the value of parameter l, each responsible peer may perform multiple expansions by itself. The states are stored in the list *targetStates* (lines 21-24). Next, for each target state, peer p checks whether it has associated queries that are successfully matched and notifies interested subscribers (lines 25-28). The target states are pushed on the runtime stack and become the active states for the next iteration of the execution (line 29). If the next parsing event is an *EndElement* event, the stack is popped (lines 30-31). Execution continues until the document has been completely parsed or the runtime stack becomes empty.

Discussion When using the iterative method, the majority of the load is imposed on the publisher peer since it is responsible for the whole execution of the NFA including contacting

Algorithm 2: `PublishIterative()`: *Publishing an XML document using iterative method*

```
1  procedure peer.PublishIterative(doc)
2      parsingEvents = parse(doc);
3      publisherId = peer.getId();
4      if l == 0 then
5          pathLength := 1;
6      else
7          pathLength := l;
8      firstPeer := Lookup("start");
9      Mesg := GetStateMesg("start") ;
10     startState := firstPeer.Route(Mesg);
11     add startState to activeStates;
12     while parsingEvents.size ! = 0 do
13         initialize event, currentEvents;
14         while currentEvents.size < pathLength do
15             event = parsingEvents.getNext() ;
16             if event is endElement then
17                 break;
18             else
19                 add event to currentEvents;
20
21         end
22         foreach state in activeStates do
23             responsiblePeer := Lookup(state.key);
24             Mesg := GetTargetStatesMesg(state, currentEvents, publisherId);
25             targetStates.add(responsiblePeer.Route(Mesg));
26         end
27         foreach state in targetStates do
28             if state.queries > 0 then
29                 notify interested subscribers;
30
31         end
32         runtimeStack.push(targetStates);
33         if parsingEvents.getNext() is endElement then
34             runtimeStack.pop();
35         activeStates := runtimeStack.getTopElement();
36     end
37 end
```

several peers to retrieve from the network the states that are not locally stored. We expect that the iterative method will perform poorly because of this potential bottleneck and we presented it here mainly to ease the reader in understanding the details of our next method. In the iterative approach, a stack mechanism is employed for maintaining multiple active paths during NFA execution. Each active path consists of a chain of states, starting from the start state and linking it with the reached target states. The main idea of the recursive method is that these active paths can be executed in parallel by different peers, which is in fact our primary motivation for choosing an NFA-based model in FoXtrot.

4.3.2 Recursive method

We now proceed with the description of the recursive method. The details of the *recursive method* are as follows. The publisher peer forwards the XML document to the peer responsible for the start state to initiate the execution of the NFA. The execution continues recursively, with each peer responsible for an active state continuing the execution. Note that the runtime stack is not explicitly maintained in this case, but it implicitly exists in the recursive executions of these paths. The execution of the NFA is parallelized in two cases. The first case is when the input event being processed has siblings with respect to the position of the element in the tree structure of the XML document. In this case, a different execution path will be created for each sibling event. The second case is when more than one target states result from expanding a state. Then, a different path is created for each target state, and a different peer continues the execution for each path.

Algorithm 3: `PublishRecursive()`: *Publishing an XML document using recursive method*

```
1 procedure peer.PublishRecursive(doc)
2 |   enrichedEvents := constructIndex(doc.parsingEvents);
3 |   firstPeer := Lookup("start");
4 |   currentIndex := 0;
5 |   parentIndex := -1;
6 |   Mesg := RecExpandStateMesg("start", enrichedEvents, 0, −1) ;
7 |   firstPeer.Route (Mesg);
8 end
```

Description Recursive method requires that a different execution path is created for each sibling event during the execution. To capture such structural relationships between elements in an XML document, we need to use a suitable positional representation. One such representation was introduced by Consens and Milo [26, 27] and has been used by many

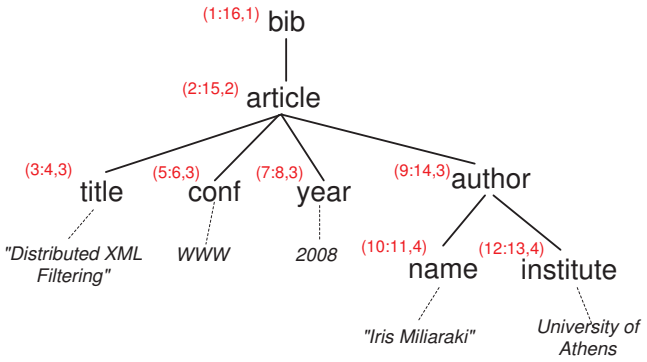


Figure 4.5: An XML document enriched with positional encoding

works for matching XML path queries like the work of Bruno et al. [16]. Specifically, the events are enriched with the position of the corresponding element with a pair (L:R,D), where L and R are generated by counting tags from the beginning of the document until the start-tag and the end-tag of this element respectively, and D is its nesting depth. Consider document nodes n_1 and n_2 and their positional encodings $(L_1 : R_1, D_1)$ and $(L_2 : R_2, D_2)$ respectively. Then, n_1 is the parent of n_2 if and only if $L_1 < L_2$, $R_2 < R_1$ and $D_1 + 1 = D_2$. The publisher peer is responsible for parsing and enriching the XML document nodes with this positional representation prior to execution. This requires a single pass over the input XML document. An example of how the nodes of an XML document are encoded is shown in Figure 4.5. For instance, the `author` node $(9 : 14, 3)$ is the parent of `institute` node $(12 : 13, 4)$, since $L_{\texttt{author}} = 9 < L_{\texttt{institute}} = 12$, $R_{\texttt{institute}} = 13 < R_{\texttt{author}} = 14$, and $D_{\texttt{author}} + 1 = D_{\texttt{institute}}$.

Algorithms 3 and 4 describe the actions required by the publisher peer and each peer responsible for an active state during recursive execution.

The publisher peer p publishes an XML document by following the steps described in procedure `PublishIterative`(doc) where doc is the XML document being published (Algorithm 3). First, peer p is responsible for enriching the parsing events using a positional representation to enable efficient checking of structural relationships (line 2). Then, peer p sends a message RecExpandStateMesg which contains the list $enrichedEvents$ with the enriched parsing events of the XML document, a pointer $currentIndex$ referring to the event that needs to be processed next (in this case 0 refers to the first element) and a pointer $parentIndex$ referring to its parent event (lines 3-7) to peer r, which is responsible for the initial state. At first, $parentIndex$ is -1 since the root element of the document has no parent element.

Let us now continue with the details of the local procedure `RecExpandState` executed at each peer that receives a RecExpandStateMesg message. The exact steps followed are

Algorithm 4: `RecExpandState()`: *Recursively expand states at each execution path*

```
1  procedure peer.RecExpandState(stateKey, enrichedEvents, currentIndex,
   parentIndex)
2      st := peer.states.get(stateKey);
3      add st to activeStates;
4      if l == 0 then pathLength := 1 else pathLength := l;
5      elementsProcessed := 0;
6      while elementsProcessed < pathLength && enrichedEvents.size != 0 do
7          currEvent = enrichedEvents.getNext();
8          if currEvent.isEndElement() then break;
9          if currEvent.hasSiblings() then
10             siblings := siblings of currEvent;
11             foreach siblingEvent in siblings do
12                 compute targetStates from each st in activeStates for input
                   siblingEvent;
13                 foreach state in targetStates do
14                     if state.queries > 0 then notify interested subscribers;
15                 end
16             end
17             break;
18         else
19             siblings := currEvent;
20             compute targetStates from each st in activeStates for input currEvent;
21             foreach state in targetStates do
22                 if state.queries > 0 then notify interested subscribers;
23             end
24         end
25         activeStates := targetStates ;
26         elementsProcessed++;
27     end
28     for i=0 tosiblings.size do
29         currEvent := siblings.get(i);
30         nextEvent := siblings.get(i+1);
31         if nextEvent is endElement then continue;
32         nextIndex := nextEvent.getIndex();
33         nextParentIndex := currEvent.getIndex();
34         foreach nextState in targetStates do
35             nextPeer = Lookup(nextState.key);
36             Mesg := RecExpandStateMesg(nextState.key, enrichedEvents,
                   nextIndex, nextParentIndex) nextPeer.Route(Mesg);
37         end
38     end
39 end
```

depicted in Algorithm 4. When peer r which is responsible for state st, receives message `RecExpandState`, it retrieves st from its local store and adds it to a list containing the active states of the execution (lines 2-3). Similar to the iterative approach, r initializes a variable called $pathLength$ using the value of parameter l (line 4). If l is equal to 0, then r can perform by itself a single expansion. Else, if l is greater than 0, r can perform multiple expansions. Peer r also keeps the number of the elements it has already processed, initially 0 (line 5). Next, r begins the execution of the NFA in the relevant path starting with state st until either it performs the relevant number of expansions or it reaches the end of the document filtered. In case the corresponding element has no siblings, r computes the expansions by itself in this single execution path and notifies the interested subscribers (lines 16-20). If the element has siblings then it computes separately the expansions for each different sibling (lines 6-15). Suppose $e_1, \ldots, e_s$ are the sibling events and $TS(e_1), \ldots, TS(e_s)$ represent the sets with the target states computed by each event. These target states may have been computed either after a single expansion or after multiple expansions. Then, r will forward $\sum \mid TS(e_i) \mid$ different RecExpandStateMesg messages, one for each of the different execution paths (lines 23-31). The execution for each path continues until the document fragment has been completely parsed. Peers that participate in the execution process are responsible for notifying the subscribers of the satisfied queries.

Note that the recursive method assumes that the XML document being filtered is relatively small and this is the reason for deciding to forward the whole document at each step of execution. In realistic scenarios XML documents are usually small as discussed in the study of Barbosa et al. [11]. However, in the case we want to filter larger XML documents, our method can be easily adjusted so that we forward smaller fragments of the document.

4.3.3 Example

An example of how peers communicate during NFA execution when using the iterative and the recursive method respectively is depicted in Figure 4.6. Peer $P10$ is the publisher of an XML document. In the case of the iterative, all communications are initiated by the publisher peer $P10$ which contacts 7 different peers for retrieving the corresponding states that are stored by those peers. When using the recursive method, execution begins with peer $P10$ contacting peer $P3$ which is responsible for the initial state. Peer $P3$ continues execution by forwarding the corresponding filtering requests to peers $P5$ and $P9$. Then, peers $P5$ and $P9$ can continue filtering in parallel as Figure 4.6 illustrates. We omit the details of the execution of this specific example and only demonstrate the sequence of the different communications occurring among the peers so that the reader can have a better understanding of the methods. However the example used is the one illustrated in Figure

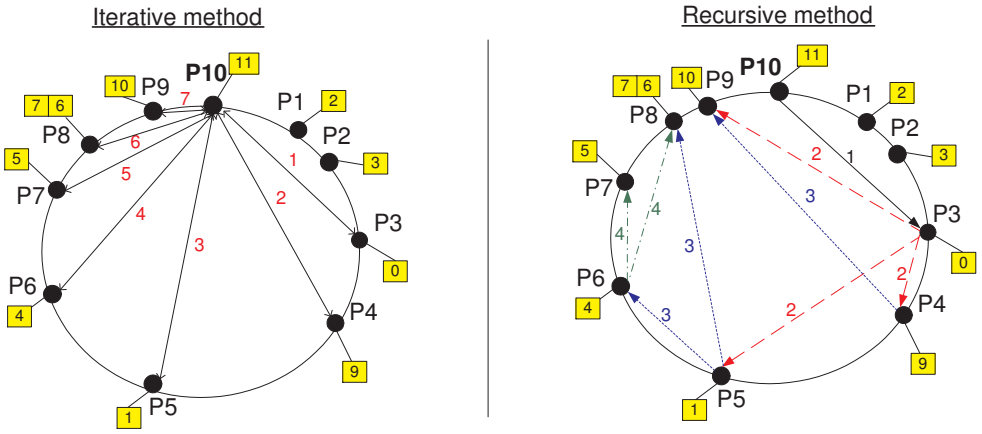


Figure 4.6: Executing the distributed NFA

4.4 for the case of the centralized NFA.

4.4 Load balancing

A core issue that arises in a distributed filtering system like FoXtrot is to have peers equally sharing the load. This is important because if a fraction of peers becomes overloaded, the overall performance of the system can be deteriorated. As we stressed earlier, apart from parameter l, which adds redundancy to our system, our distribution scheme stores the start state at a single peer creating a potential bottleneck. For this reason, we employ replication techniques to achieve a more balanced distribution of the load in FoXtrot. In this section, we describe the *load balancing* techniques we have designed and developed in FoXtrot.

4.4.1 Overview

In systems like ONYX [29], it is required to adopt a strategy for deciding where to store queries and where to deliver XML data. For example, the authors in ONYX use criteria like the topological distance between the broker and the data source, the available bandwidth, the content of the query, and the location of the subscriber. This selection process is performed by a centralized component. Instead, FoXtrot uses a single NFA to index the queries and this NFA is distributed among the peers using the DHT mechanism provided by Pastry. As a result, NFA fragments are assigned to peers in a random way. This leads to a fairly uniform distribution of storage load among the network peers without requiring any additional actions.

However, even when peers share equally the fragments of the NFA (i.e., storage load is evenly distributed), filtering load distribution can be very unbalanced due to the following

reasons. Firstly, given that the distributed NFA is a tree-like structure, this causes peers responsible for the states of smaller depths to suffer more load than the others. Consider for example the peer responsible for the start state. Secondly, the distribution of element names in the XML document set being filtered can be skewed and the relevant states will be accessed more frequently. The same holds for the distribution of element names in the query set. We are mainly concerned with the balancing of the filtering task since this is considered the heaviest. Each peer which receives a filtering request in FoXtrot, retrieves the relevant NFA states, performs execution, dispatches notifications if queries are matched and if execution has not ended, creates and forwards a new filtering request.

In the following, we first describe the load balancing methods we employ, namely *static* and *dynamic replication* and later in the section we demonstrate their efficiency experimentally. Apart from the methods designed for FoXtrot, we also describe two generic techniques that we employed for our initial simulations, presented in [66], namely *virtual nodes* and *load-shedding*. In general, there is a lot of work that studies efficient load balancing methods in DHTs [52, 95, 100]. Note that we design our methods assuming a network consisting of peers with similar capabilities and our goal is to evenly distribute the load among them. We do not consider the case where the network consists of a heterogeneous group of peers.

4.4.2 Static replication

Given our state distribution technique, there is one responsible peer for each state. As a result, the states that are more frequently accessed will cause the relevant peers to suffer more load. Such examples include states of smaller depths and also states accessed due to a skewed distribution of element names as mentioned previously. Consider for example a bibliographic database where a user wants to be notified when a certain author publishes an article. The corresponding query is q: $/bib/article/author[text() = authorname]$. This is a common case and the state of the relevant path would be accessed more frequently causing a potential bottleneck for our system. Along with our state distribution technique, we design replication techniques for storing states multiple times across the network. Increasing parameter l also affects the distribution of load since peers are able to perform execution at a larger NFA fragment but this is not sufficient for achieving a uniform load distribution.

Our first method creates a fixed number of r replicas, where r is called the replication factor, for each NFA state. Instead of having a single peer responsible for a state, we assign the same state to r responsible peers. We refer to this method as *static replication*. Replication is performed during query indexing and whenever a peer creates a state for which it is responsible, it also creates r replicas for this state and randomly selects the peers to store them. This is accomplished as follows. Recall that each NFA state is identified by a

key k. The responsible peer for state with key k is the peer whose identifier is numerically closest to $Hash(k)$, where $Hash()$ is the DHT hash function. Now, in order to create r replicas for state st, instead of indexing st only according to k we also index it using the keys $k_1 = k+1$, $k_2 = k+2$, ..., $k_r = k+r$. Operator $+$ is used to denote the concatenation of strings. These correspond to the *replication keys* and lead to the peers responsible for the replicated states. During filtering, if a peer wants to forward a request for state st, it will choose randomly among the r peers and the load that would be suffered by a single peer is now distributed among the $r+1$ peers.

An obvious drawback of static replication is the extra storage overhead suffered by the peers as we increase the replication factor. Even if the storage overhead is considered negligible, it causes an increased latency during indexing since r times more states need to be created and stored. We will study this in detail during our evaluation.

4.4.3 Dynamic replication

To avoid the excessive storage requirements of static replication, which can cause latencies during indexing, we improve our method as follows. We assume that the frequency of visiting an NFA state during filtering is inversely proportional to the depth of this state. This assumption is made having in mind that the tree structure of the NFA is the main reason causing the load imbalances (e.g., if $r > 0$, one of the r peers responsible for the start state will receive a filtering request each time an XML document arrives at the system). For this reason we create a different number of replicas for each state depending on its NFA depth. So, instead of having a fixed number of replicas for each NFA state, we create a number of r/d replicas for each NFA state of depth d. For the purposes of this method, we consider that the depth of the start state is 1, its children states have depth 2, and so on. We refer to this method as *dynamic replication*.

Another interesting case is when the frequencies of visiting the NFA states are not dependent on the depth of the states but follow a different distribution. In this case, the number of the replicas for a state should be proportional to its access frequency f. Estimating these frequencies is an interesting problem which we have not considered.

4.4.4 Virtual nodes

There have been many DHT proposals including Chord [93], CAN [85] and Pastry [88]. Even though these different systems are based on the same concept of employing a hash function for distributing data items among the network peers, the exact distribution differs depending on the specific overlay. In the case of Chord, where consistent hashing is used,

with high probability for any set of N nodes and K keys, each node is responsible for at most $(1+\epsilon)K/N$ keys. Karger et al. [51] show in their paper regarding consistent hashing that ϵ can be reduced to an arbitrarily small constant by having each node run $O(\log N)$ "virtual nodes" each with its own identifier. As a result, the load balancing scheme proposed called *virtual nodes* balances the number of keys per node by associating keys with *virtual nodes* and then mapping multiple virtual nodes (with unrelated identifiers) to each real peer. As claimed by Karger et al. [51] and demonstrated in Chord paper [93], we need to allocate $logN$ randomly chosen virtual nodes to each peer for ensuring an equal partitioning of the identifier space between the peers.

4.4.5 Load-shedding

Another popular method for achieving load balancing in distributed systems and DHTs more particular, is based on the concept of *load-shedding*. Such a method has been used successfully in works like the one of Galanis et al. [36] and also by Tryfonopoulos [100] who study information filtering in a DHT environment. The main idea is that when a peer p becomes overloaded, it chooses the most frequently accessed state st and contacts a number of peers requesting to replicate state st. Then, p notifies the rest of the peers that st has been replicated, so that if a peer needs to retrieve it, it will randomly select one of the responsible peers.

4.5 Fault-tolerance

As the size of distributed systems increases daily, the chances for failures are higher. Examples of failures include failures of processors, disks, memory, power and network link failures. A large-scale distributed system like FoXtrot, expected to run on collections of machines on the Internet, should be designed to operate in the presence of failures. In this section, we discuss ways to increase the resilience of FoXtrot to failures.

4.5.1 Overview

In a distributed system, a node may depart without prior warning due to a network failure or even leave the network at its own will. We are mainly concerned with the former case where a departure of a peer is sudden. Fault tolerance has been studied extensively for the case of distributed systems in general [28] and also more specifically for peer-to-peer environments [8, 59, 80].

We discussed earlier some replication-based techniques for improving load distribution in

FoXtrot. Replication-based techniques, apart from assisting us in the task of load balancing, they can also increase the fault tolerance of the system through introducing redundancy.

4.5.2 Techniques

In general, by having a replication degree r (i.e., the number of replicas is r) a system can tolerate up to $r-1$ faults. In other words, if at most $r-1$ peers fail simultaneously, this will not affect the system and thus fault tolerance is achieved in this case. Hence, by using static replication with r replicas, FoXtrot can tolerate up to $r-1$ faults. To achieve this, each peer p who forwards a request for a state st to another peer will wait to get an acknowledgement. If it fails to get this acknowledgement, it redirects the request to a peer which owns a replica for state st, and so on. In case, r or more peers fail, then fault tolerance is not guaranteed. Of course, a larger number of replicas also means a larger overhead for the system in terms of creating and storing these replicas.

Besides explicitly using a replication-based technique, recall that our distribution scheme in FoXtrot uses parameter l and allows peers to store additional states (i.e., apart from those for which they are responsible) so that each peer is able to execute a larger part of the NFA during filtering. This also adds redundancy to FoXtrot and can also be exploited to increase the resilience of the system to failures. Let us describe how exactly we achieve this. Consider a peer p who forwards a request for a state st to another peer. If it fails to get an acknowledgement and $l > 0$, it redirects the request to the peer which is responsible for the parent state of st, st_{parent}. Due to l, this peer will not only keep locally state st_{parent} but also all the states reachable from st_{parent} after following a path of length l. Since $l > 0$ this path includes state st. Now, if $l > 1$, then the peer can also contact the peer responsible for the parent of state st_{parent} and so on.

Example 4.5.1 (Using parameter l for fault-tolerance)**.** *Let us demonstrate the above using an example. Figure 4.7 illustrates a small NFA fragment consisting of* 4 *different states. We also show the peers responsible for each of the states depicted in the figure. As we increase l, states become replicated across the different peers. If $l = 0$, the failure of a single peer will cause loss of the states stored locally at the peer. So as expected when $l = 0$, FoXtrot exhibits zero tolerance to failures. For a larger value of $l = 2$, if only a single peer fails, then the system can continue operating under normal conditions. However, if more than one peer fails, fault tolerance is not guaranteed.*

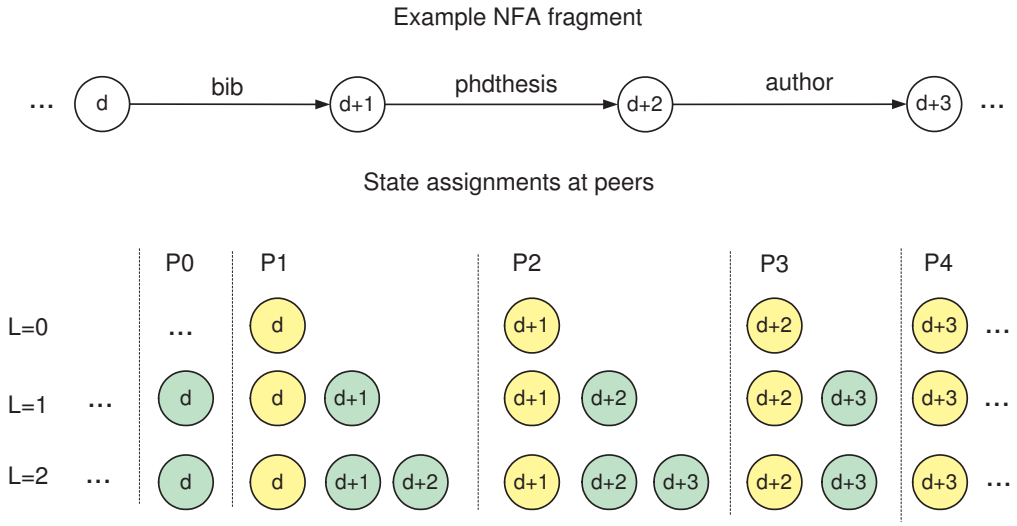


Figure 4.7: State replication due to parameter l

4.6 Experimental evaluation

In this section, we study the performance of FoXtrot focusing on structural matching. We begin by describing our experimental settings, the datasets used for our evaluation and the metrics that interests us with respect to the performance of FoXtrot. Then, we give a detailed description of our evaluation and demonstrate the performance of FoXtrot under various scenarios. Finally, we discuss our results and compare them with other related approaches.

We point out that there are cases where it is interesting to also include predicates in the queries. In such cases, predicate evaluation takes place after the execution of the NFA. However, we do not provide any more details at this point about how this is achieved since this is the topic of the next chapter, where different value matching methods are described and evaluated.

4.6.1 Setup

Our structural matching methods were initially evaluated [66] using a simulated Chord network [93] we developed in Java. We have also implemented FoXtrot in Java using FreePastry release [35]. In the latter case, we run our experiments in two different environments, the worldwide testbed for large-scale distributed systems provided by the PlanetLab network[1] which represents the real-world conditions of the Internet and a local shared cluster[2]. As expected, during our evaluation, we mainly focus on the experiments conducted using the

[1]PlanetLab global research network `http://www.planet-lab.org/`
[2]Grid computer at Technical University of Crete `http://www.grid.tuc.gr/`

implementation of FoXtrot but we also report some interesting simulation results.

Network setup In the case of PlanetLab, we used 396 nodes that were available and lightly loaded at the time of the experiments. Note that PlanetLab nodes are geographically distributed among four continents and shared by many users. We also ran our experiments on a cluster that consists of 41 computing nodes. Each node is a server blade machine with two processors at 2.6GHz and 4GB memory. In this case, we used 28 of these machines running up to 4 peers per machine, i.e., 112 peers in total. With respect to the experiments we ran using our Chord simulator, we created networks of 10^3 to $5 * 10^3$ peers.

XML documents We generated many different synthetic data sets using the IBM XML generator [46] and also used a real dataset consisting of the DBLP XML records [1]. Each set included 1000 documents. Our synthetic datasets were created using the NITF (News Industry Text Format) DTD and the Auction DTD from the XMark benchmark [106]. We also created a mixed dataset using a set of 10 different DTDs. Using the mixed dataset, we study the performance of our approach in a realistic scenario where users subscribe to FoXtrot to receive notifications concerning various interests of theirs (e.g., information about scientific papers and news feeds). The NITF DTD has been used in many works [20, 30, 44] for evaluating XML filtering functionality and represents an interesting case since it allows 123 different element tags, 513 attributes and a large fraction of XML elements can be recursive. On the other hand, the DBLP DTD represents a quite simplistic case including 36 different element tags and 14 different attributes. If not explicitly mentioned, our measurements presented in this chapter concern the NITF dataset. However, in cases where the performance of FoXtrot is affected by the corresponding dataset, we also include the relevant experiments. The values of the parameters used for generating the document sets are shown in Table 4.2.

XPath queries The same DTDs were used to generate different sets of 10^6 path queries with varying characteristics using the XPath generator available in the YFilter release [107]. Each query set contained *only distinct* queries, in other words there are no duplicates. In a realistic scenario where users share interests, such a query set can represent the interests of millions of users. The values of the parameters used for generating our query sets are shown in Table 4.2.

Evaluation metrics We are mainly interested in measuring the time spent and the network traffic generated during indexing XPath queries and filtering XML data. We also study how this traffic is distributed among the network peers. More formally, the metrics used in

Parameter	Default	Range
Number of documents	10^2	$10 - 10^3$
Document depth	10	$5 - 25$
Number of queries	10^6	$10^5 - 10^6$
Query depth	12	$5 - 15$
Predicates per query	2	$1 - 3$
Wildcard probability	0.2	0.2
Descendant axis probability	0.2	0.2
Skewness of element names (θ)	0	0

Table 4.2: Dataset generation

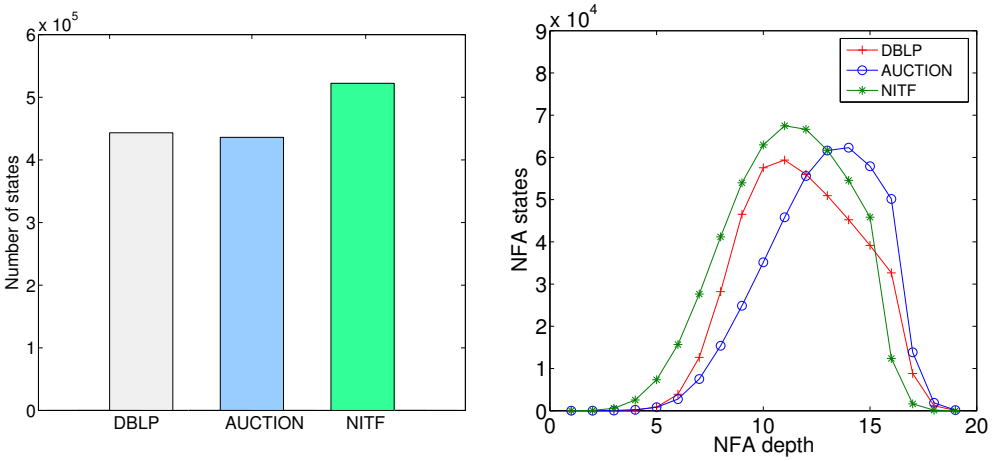
Parameter	Default
Network size (Cluster)	112
Network size (PlanetLab)	396
Structural matching	Recursive method
Value matching	Succeeds structural
Parameter l	2

Table 4.3: FoXtrot setup

the experiments as well as their definitions are the following. The *indexing latency* for a set of queries Q is measured as the amount of time spent until all queries of Q are indexed in the system. *Indexing throughput* is measured as the number of queries indexed over a specified time period. The *filtering latency* for a set of XML documents D is measured as the amount of time spent until all notifications are dispatched to the interested subscribers for the queries matched by the documents of D. The *network traffic* is measured as the total number of messages generated by network peers during indexing queries and filtering incoming XML data. We also distinguish the following types of peer load. The *filtering load* of a peer is measured as the total number of messages a peer sends during a filtering operation, while the *storage load* of a peer is measured as the total number of states it stores locally. Finally, we will use the term *NFA size* to refer to the total number of states included in the distributed NFA that is shared by the peers.

FoXtrot setup To carry out our experiments we execute the following steps. We create a network of n peers connected using Pastry DHT and implementing the functionality of FoXtrot. Then, we index a set of queries Q in the system using randomly selected peers as subscribers and study the performance of FoXtrot with respect to the metrics described above. Last, we filter a set of XML documents D, using random peers as publishers, and measure again all relevant metrics.

Our default method for executing the distributed automaton is the recursive method which as expected performs better than the iterative method as we also demonstrate in our experimental evaluation. Table 4.3 summarizes the default values for setting up FoXtrot including the value of parameter l.



(a) NFA size (b) NFA states per depth

Figure 4.8: Distributed NFA characteristics

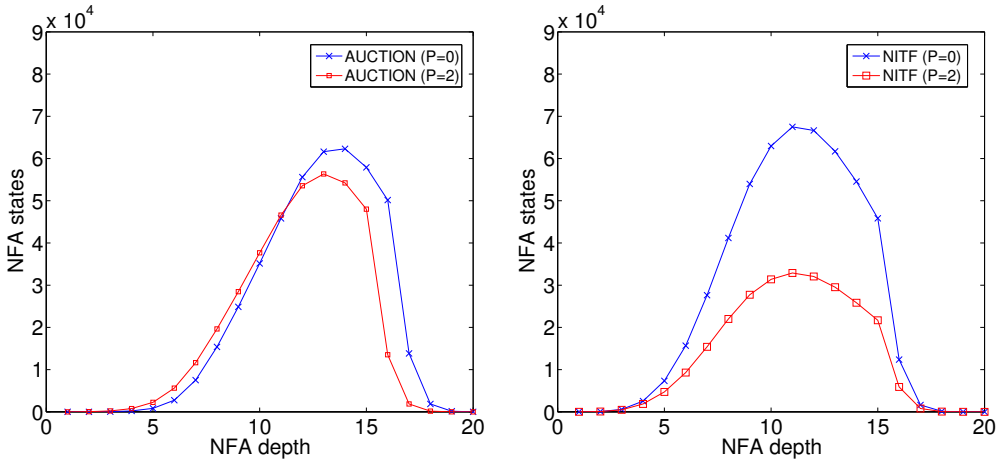
4.6.2 Results

In this section, we present our results. Our main goal is to demonstrate the scalability of FoXtrot and its load balancing properties under various conditions including a very large set of queries and a high rate of incoming data. The following experiments are divided into four groups. We begin our evaluation by studying the properties of the distributed NFA shared among the peers. Then, we study how load balancing techniques affect the distribution of the load imposed on the network peers. In the third group, we study the performance of FoXtrot during query indexing. Then, in the fourth group of experiments, we demonstrate how FoXtrot operates during XML filtering. Finally, we summarize our evaluation by discussing our results. Unless otherwise stated, our results are obtained by running the experiments on the cluster. In cases where we observed differences among the experiments in the two environments, we point out these differences and discuss them in detail. We also include in some cases our simulation results from our initial evaluation [66].

4.6.2.1 Distributed NFA properties

In this group of experiments, we study some properties of the distributed NFA shared among the network peers in FoXtrot. In particular, we focus on the size of the NFA measured in states and how these states are distributed among the different levels of the NFA.

We begin by demonstrating how the structure of the NFA differs for the different query sets we consider. The main characteristics of these DTDs are shown in Table 4.4. In general, the number of the NFA states depends on the properties of the relative DTD and the characteristics of the query set. A larger number of elements allowed in a DTD results



(a) NITF query set (b) Auction query set

Figure 4.9: NFA states per depth

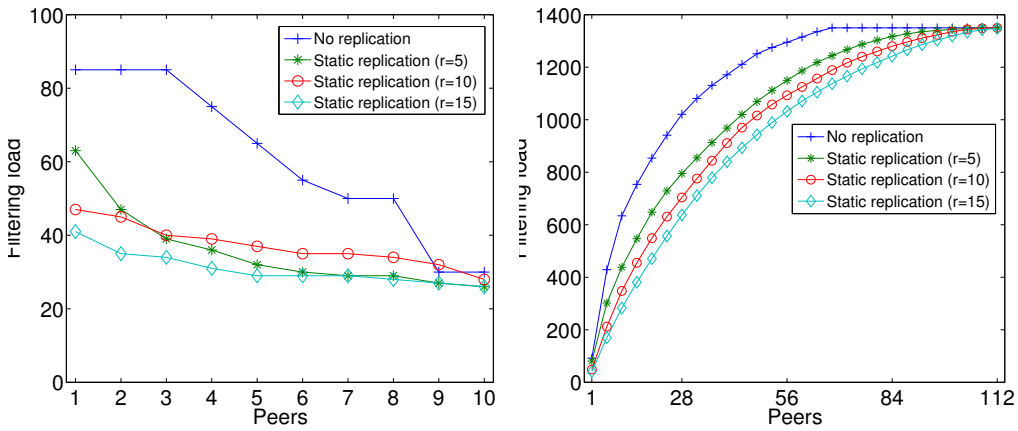
in a broader NFA (greater branching factor for each state), while a larger recursion level increases the depth of the NFA. For the generation of the queries we allow a maximum query depth of 15 steps. Note that since the NFA also includes ϵ-transitions to represent descendant axes, the length of the NFA can be larger.

	DBLP	**AUCTION**	**NITF**
Number of different elements	36	77	123
Number of different attributes	14	16	510
Larger recursion level	infinite	infinite	infinite

Table 4.4: DTD characteristics

Figure 4.8 illustrates the size of the NFA and how the states are distributed among the different NFA levels for the NITF, Auction and DBLP query sets respectively. Each query set in this experiment consists of 10^5 distinct queries with no predicates. In the case of the NITF query set, we observe that the corresponding NFA is the largest containing about $9 * 10^4$ more states than the others. This is expected as the NITF DTD allows a significantly larger number of different elements.

In Figure 4.8(b) we show how the states are distributed among the different NFA levels for each query set. Again, in the case of NITF query set the increase rate of the states at each NFA level is higher than the rate observed in the other query sets. As explained in the previous chapter, we utilize the XPath generator from YFilter [107] for constructing our query sets. It is interesting to study how the different characteristics of the queries affect the structure of the NFA. Recall that we only include queries distinct from each other. In this experiment, we demonstrate how the structure of the NFA differs as we increase the number of predicates included in each indexed query. The results are shown in Figures 4.9(a)



(a) Most loaded peers (b) Filtering load (all peers)

Figure 4.10: Load distribution using static replication

and 4.9(b) for the NITF and the Auction query set. In the case of Auction DTD, the size of the NFA as we increase the number of predicates per query to 2 is slightly decreased but the overall distribution of states is similar. A more interesting example is the NITF query set. As Figure 4.9(b) depicts, the set containing queries with predicates corresponds to a significantly smaller NFA and a more balanced distribution of states across the NFA structure. The main reason is that the large number of attributes defined in the NITF DTD (i.e., 510 different attributes) allows to create a large number of distinct queries sharing the same structural path.

4.6.2.2 Load balancing

In the following experiments, we evaluate our load balancing methods, namely static and dynamic replication, using the following steps. We create a network of 112 peers, index $5 * 10^5$ path queries, and publish repeatedly 100 XML documents simultaneously using random peers as publishers.

Static replication We begin with demonstrating how static replication affects load distribution in FoXtrot. The results, when we vary the number of replicas r from 0 to 15, are presented in Figure 4.10. First, in Figure 4.10(a), we show the 10 peers that suffer the most load in a descending order of filtering load. As we can see, when no replication is used, a fraction of peers is overloaded receiving a large number of requests, while other peers receive only a small proportion of the total load. Even when a small number of replicas is created in FoXtrot, load distribution is considerably improved. When 15 replicas are created, load distribution is further improved having the 10 most loaded peers receiving almost equal loads

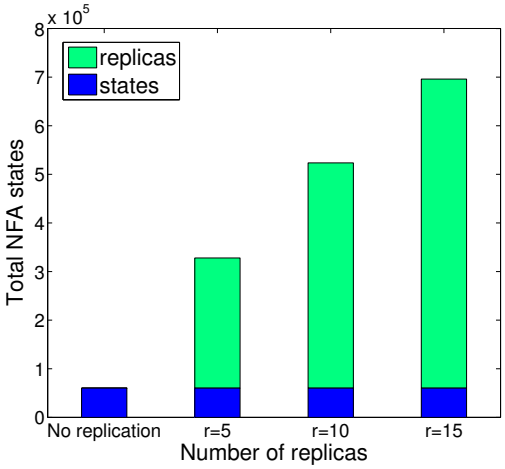
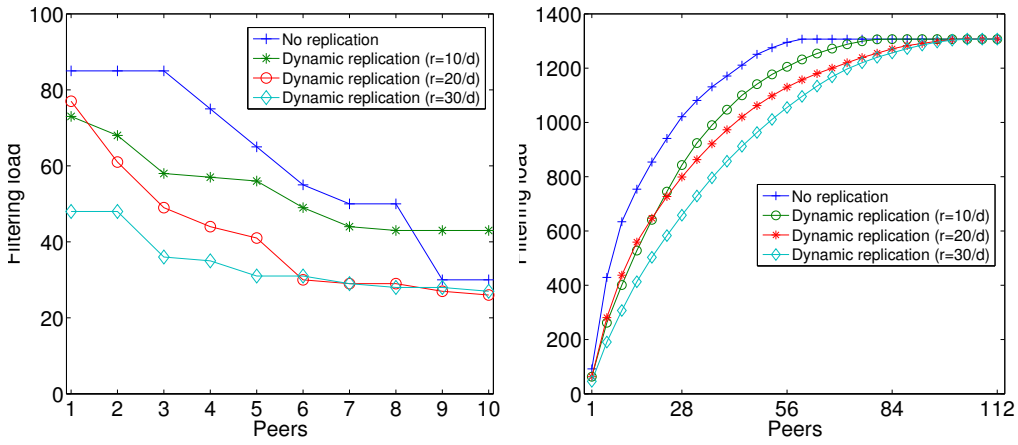


Figure 4.11: Storage overhead for static replication

and eliminating potential bottlenecks.

In Figure 4.10(a), we concentrated on the part of the network that receives the most load, disregarding the overall load distribution. In Figure 4.10(b) we show the distribution of load among all the network peers. On the x-axis, peers are ranked starting from the peer with the most filtering load. The y-axis represents the cumulative filtering load, i.e., each point (x,y) in the graph represents the sum of filtering load y for the x most loaded peer. When no replication is used, the filtering load is very unbalanced and many peers receive very few or no requests at all. Particularly, more than 40 peers do not receive any filtering request from the total 1350 requests that are generated during filtering in FoXtrot (see the straight line segment when $x > 70$). By using replication, we quickly observe a more even distribution of load which improves as we increase replication factor from 5 to 15. Also all the networks peers participate in the filtering process. We also measured the variation of the different peer loads using the metric of standard deviation (σ) and we observed that deviation is decreased as we increase the number of replicas for each NFA state. In other words, less dispersion is observed among the different peer loads. For example, when no replication is used $\sigma \simeq 20$, while when $r = 15$, $\sigma \simeq 8$.

However, the price we pay for a more uniform distribution of the load is the large storage overhead suffered by the peers as we increase the total number of replicas. We are mainly concerned with the creation of this large number of replicas because it can deteriorate indexing performance in terms of latency since actual storage costs even for a large number of states are negligible (measured in MBs). We demonstrate in Figure 4.11 how storage load increases as we increase the number of replicas. As expected, the number of replicas is high and as an illustration, when $r = 15$, the storage overhead is more than $6 * 10^5$ replica states. Note that storage load includes some redundant states because of parameter l as discussed



(a) Most loaded peers

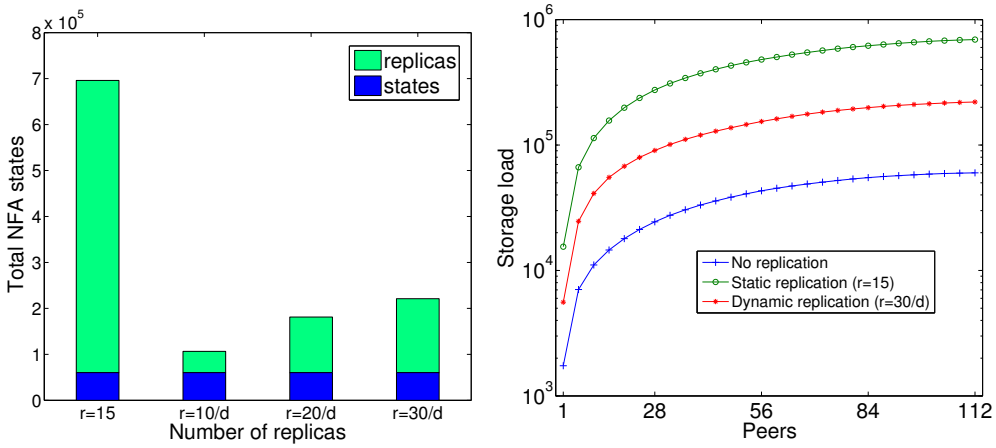
(b) Filtering load (all peers)

Figure 4.12: Load distribution using dynamic replication

in Section 4.1. However we do not create replicas for these states and that is the reason that a replica factor r results in less than r times the number of states as shown in Figure 4.11.

Dynamic replication We now continue with the evaluation of dynamic replication method. As we have explained, dynamic replication avoids the excessive storage requirements of static replication by creating a different number of replicas for each NFA state of depth d. We run the same experiments as before to evaluate the dynamic replication method and the results are presented in Figure 4.12. We first demonstrate how load is distributed among the 10 peers that suffer most of the load. In Figure 4.12(a) on the x-axis peers are ranked starting from the peer with the most filtering load. We observe that as we increase the replication factor the peer that receives the most filtering requests suffers less load. At the same time load is distributed in a more uniform way among the other peers. Static and dynamic replication techniques exhibit a similar performance when $r = 15$ and $r = 30/d$ respectively (see also Figure 4.10(a)). The main advantage of dynamic replication is that we achieve this while keeping storage overhead low. As Figure 4.13(a) shows, a replication factor of $30/d$ almost triples the NFA states stored by the peers. This compares favorably with static replication where we achieve a similar load distribution for the case of 15 replicas (see Figure 4.10(b)), but the resulting amount of storage load was 9 times the number of states (see first bar of Figure 4.13(a)).

We also demonstrate the overall load distribution in Figure 4.12(b). As previously, on the x-axis peers are ranked starting from the peer with the most filtering load and y-axis represents the cumulative filtering load. Creating a varying number of replicas depending on the depth of each NFA state, results in a more even distribution of load which improves



(a) Storage overhead for dynamic replication (b) Storage load distribution

Figure 4.13: Storage overhead and storage load distribution

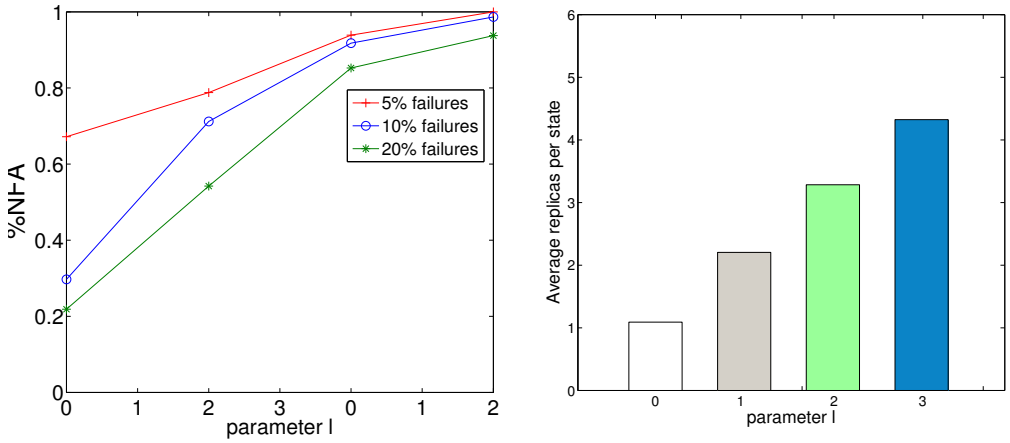
as we increase the replication factor (from $r = 10/d$ to $r = 30/d$). When we use dynamic replication, all peers participate in the filtering process. We also measured the variation of the different peer loads using standard deviation and we observed that deviation is relatively low. For instance when $r = 30/d$, $\sigma \approx 10$ (the total number of filtering load is 1350 requests).

Storage load For completeness we also demonstrate in this group of experiments the storage load distribution in FoXtrot. The results are shown in Figure 4.13(b). We plot our results on a logarithmic scale since the total storage load differs considerably among the different load balancing techniques. Again y-axis represents the cumulative load with peers ranked on x-axis in a descending order of their load. We can see in Figure 4.13(b) that even when no replication is used, as expected, storage load is distributed in a fairly uniform way among the peers due to the randomness of our distribution method. We report that a small group of peers stores a larger fraction of the total states, however in case of storage load, as we explained previously, these differences can be considered negligible (measured in MBs). Also, increasing the number of replicas slightly improves storage load distribution.

4.6.2.3 Fault tolerance

As we discussed earlier, our replication-based methods and parameter l adds redundancy to our system and as a result increases the fault tolerance of FoXtrot. In this set of experiments, we study how exactly our system operates under failures.

First, we study the impact of parameter l without creating any more replicas. Our experiment is conducted as follows. We create a network of 100 peers and index 10^5 queries. Then, we randomly select n peers to disconnect from the network simultaneously. We analyze



(a) Percentage of accessible states (b) Storage overhead

Figure 4.14: Fault tolerance (parameter l)

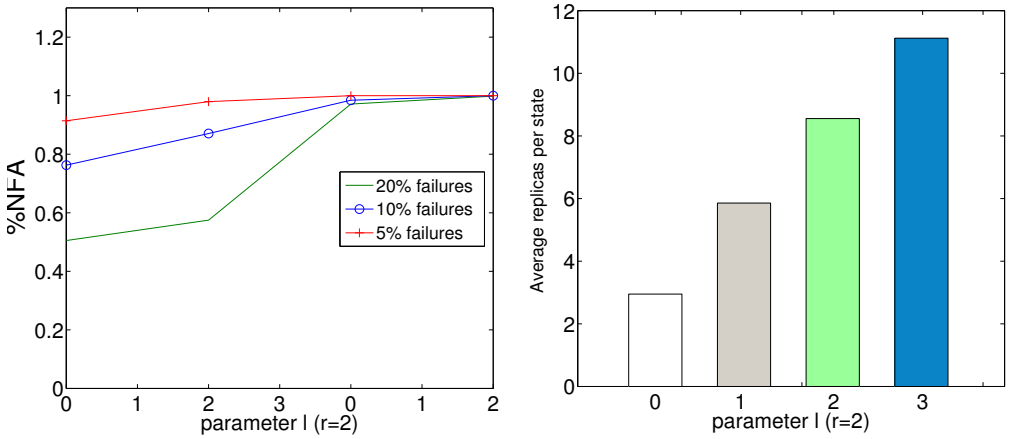
the impact of the failures by attempting to traverse the entire distributed NFA beginning from the initial state and following transitions. We report the percentage of the total NFA states that we were able to access. Note that if a state becomes inaccessible after the failure of all peers which kept it, we fail to traverse all the states that follow. We repeat this experiment for $n = 5$, 10, and 20 and the results are shown in Figure 4.14 as we increase parameter l.

As we can see from Figure 4.14, when $l = 0$ and each state is stored at single peer, a large part of the NFA cannot be accessed. In particular, when 10 peers fail only 30% of the NFA structure can be accessed. As we increase parameter l, the resilience of the system to failures is improved and when $l = 2$ even with 20 peers failing more than 80% of the NFA can be traversed. We also depict the storage overhead from the additional states created due to parameter l in Figure 4.14(b).

We also performed experiments combining l with our replication-based methods. The results when $r = 2$ are shown in Figure 4.15(a). We observe that FoXtrot becomes less sensitive to failures, exhibiting high resilience to failures when 5% of the network peers fail. However, there is a higher storage overhead due to the creation of a larger number of replicas depicted in Figure 4.15(b).

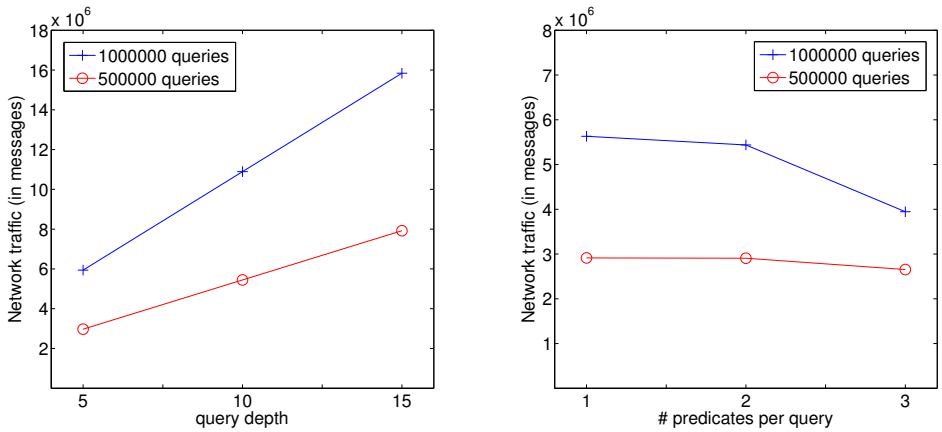
4.6.2.4 Indexing queries

In this section we demonstrate how FoXtrot performs during query indexing. We are mainly interested in the number of messages that travel through the network and the time spent when indexing a set of queries. We study how the performance of the system is affected by the different characteristics of the query set (e.g., query depth and number of predicates per



(a) Percentage of accessible states (b) Storage overhead

Figure 4.15: Fault tolerance (parameter l)



(a) Varying query depth (b) Varying predicates per query

Figure 4.16: Network traffic during query indexing

query).

Network traffic In this group of experiments we study the network traffic that is generated during query indexing. We begin with examining the impact of query depth on the generated traffic and continue with how the number of predicates per query affects the network traffic.

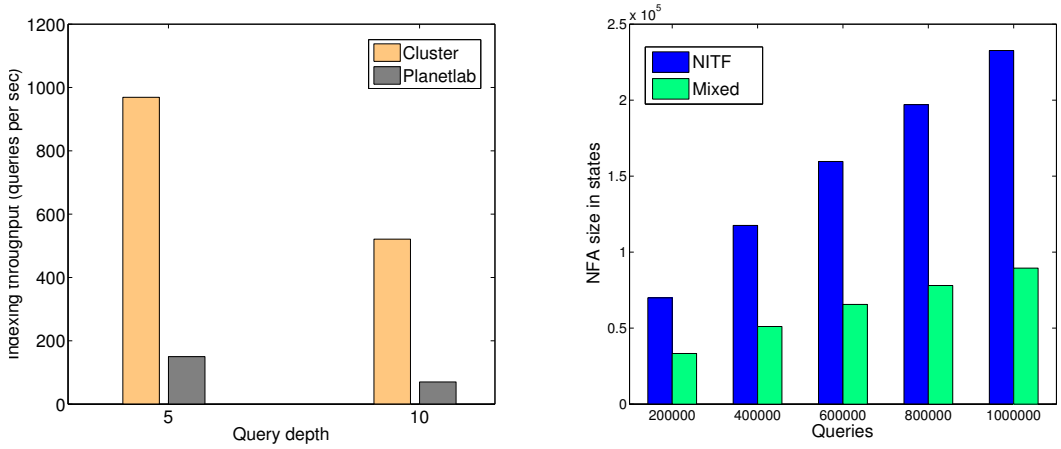
We create a network of 112 peers and index three different query sets containing queries with depths 5, 10, and 15 respectively. The results are shown in Figure 4.16(a) for the cases where $5 * 10^5$ and 10^6 queries are indexed in FoXtrot. The graph shows the total amount of network traffic generated during the indexing of queries. In both cases, as Figure 4.16(a) depicts, the network traffic generated scales linearly with the depth of the queries being indexed. Particularly, for the case of 10^6 indexed queries, as we increase the query depth

from 5 to 15, FoXtrot generates from $6 * 10^6$ to $16 * 10^6$ messages respectively. This is due to the fact that indexing a single query of depth d requires sending at most $d + 1$ messages, i.e., one message to the peer responsible for the start state and d additional messages to the peers responsible for the other d states. These messages either update or create the corresponding NFA states. So, for indexing a set of 10^6 queries consisting of 5 steps each, at most $6 * 10^6$ messages travel through the network. In some cases the messages actually sent may be slightly less since peers can be responsible for subsequent states and less than $d + 1$ messages are needed for a query of depth d. Note that for the purposes of this experiment, we prefer to index queries one at a time in each iteration. However, if queries arrive in chunks, we can decrease the number of messages by performing a bulk indexing operation for each chunk instead of several separate operations.

Apart from query depth, we also examine how the number of predicates per query affects network traffic during indexing. In this case instead of increasing the depth of each query, we increase the number of predicates per query. Again, Figure 4.16(b) shows the total amount of network traffic generated during indexing query sets with 1, 2, and 3 predicates respectively. We present two cases when $5 * 10^5$ and 10^6 queries are indexed in FoXtrot. While network traffic increases linearly with the query depth, the total number of predicates included in each query does not significantly affect the number of indexing messages sent in all cases. We observe a decrease of network traffic as the number of predicates per query is increased. However, this is actually caused by the method we use for synthetically generating our queries. As we increase the number of predicates allowed per query, the query generator creates a set where queries share more structural similarities. In other words, the distributed NFA that is constructed is smaller and as a result less messages travel through the network for traversing it during indexing. This is depicted more clearly in the case of 10^6 queries where we observe a 30% decrease on network traffic as we increase the predicates to 3 per query (average query depth in this case is 6).

Let us now study the throughput of FoXtrot during query indexing. We measure throughput as the number of queries indexed in a given amount of time. While network traffic measurements are obviously unaffected by whether we run our experiments using the PlanetLab network or the cluster, indexing throughput is considerably different. The main reason is that network delays in a setting like PlanetLab, where peers are geographically dispersed, are significantly higher compared to the ones observed in the cluster.

Before proceeding with the results, we describe a cache mechanism we used for improving the performance of FoXtrot and decrease latencies. During query indexing we repeatedly visit the same states of the distributed NFA by contacting the relevant peers. We take advantage of this and cache useful routing information at each peer. Consider for example a peer p which is responsible for a state st. Each time another peer p' wants to forward an



(a) Indexing throughput (b) NFA size

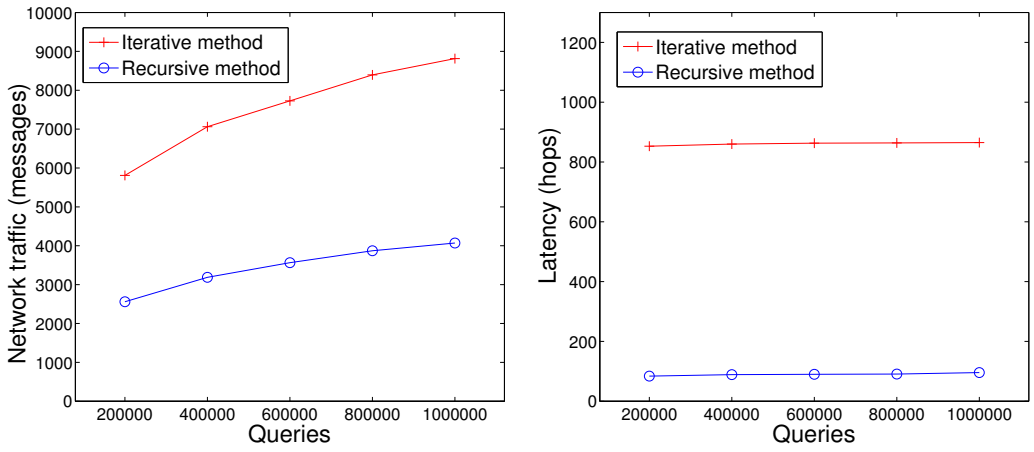
Figure 4.17: Indexing operation (II)

indexing message to p as responsible for state st, the message will travel $O(\log n)$ hops to reach its destination.[3] We can avoid this by having p' keeping the IP address of p as the peer responsible for state st. So, if p' wants to contact again the peer responsible for state st, it will first check its local cache and then the message will be delivered in a single hop. Such a caching technique is standard in these settings [100, 57] and helps to reduce latency since messages reach faster their destinations.

Indexing throughput In Figure 4.17(a), we demonstrate the throughput achieved by FoXtrot in queries per second. In both cases, we create networks of 100 peers. In the case of PlanetLab, when query depth is 5, only 150 queries are indexed per second, while throughput drops to less than 70 queries per second when query depth is increased to 10 steps. This is due to the fact that as query depth increases, so does the indexing time, since the number of messages that are sent through the network are increased. FoXtrot exhibits a significantly better performance on the cluster reaching a throughput of 969 queries per second when queries contain 5 steps. In other words, the throughput FoXtrot achieves using the cluster machines is 1 order of magnitude higher than when we run FoXtrot on PlanetLab. We also report that in the case of PlanetLab, our measurements suffered from an increased variation. This was due to the existence of a few arbitrarily slow nodes. This problem has been studied in the context of a public DHT service called OpenDHT which was deployed on PlanetLab by [86]. The authors focus on the problem of slow nodes and demonstrate ways to overcome their effect on the performance of the system.

Again, we expect that we can further increase indexing throughput by performing a bulk

[3]Pastry routes messages to the peer whose identifier is numerically closest to the given key using prefix routing and each such request can be done in $O(\log n)$ steps, where n is the number of nodes in the network.



(a) Network traffic (b) Latency

Figure 4.18: Iterative vs. Recursive method

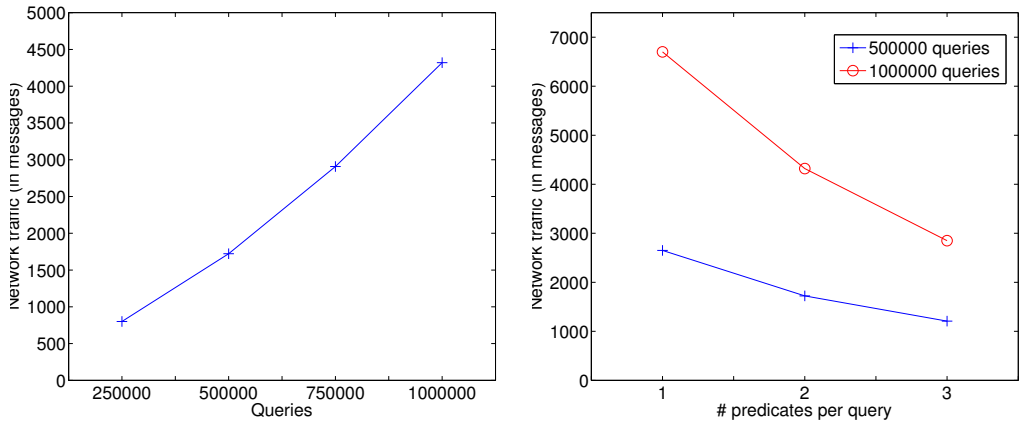
indexing operation for each chunk of queries. We expect that this would benefit more an environment like PlanetLab where network latencies are large. We do not include results for the case where we increase the number of predicates per query. The reason for this is the strong dependence of indexing latency on the number of predicates per query for which we have carried out the recent study [67].

4.6.2.5 Filtering documents

We continue our evaluation by studying the performance of FoXtrot during XML filtering. First, we compare the two methods we described for executing the distributed automaton in FoXtrot, namely the iterative and the recursive method. With respect to the evaluation of the filtering performance of our system, we are mainly interested in the number of messages that travel through the network and the time spent when filtering a set of XML documents.

Iterative vs. Recursive method We have described two methods for structural matching, namely the iterative and the recursive method. We have chosen to implement only the recursive method in FoXtrot since as expected and demonstrated through simulation results [66] it outperforms the iterative one in terms of latency since it distributes the load more evenly and generates less network traffic. Figure 4.18 presents two graphs that illustrate this clearly.

Network traffic Let us first study the number of messages that are generated during the filtering process in FoXtrot. For the purposes of the experiments, we create a network of 112 peers and incrementally index 10^6 path queries. After each indexing iteration, we publish



(a) Increasing indexed queries (b) Increasing predicates

Figure 4.19: Network traffic during filtering

the whole document set consisting of 100 XML documents and measure the network traffic generated during the filtering of these documents. We repeat these steps for various cases. Note that the notification messages generated when an XML document matches an indexed query are not considered part of the network traffic. In other words, we only consider the messages generated for the purposes of filtering.

In our first experiment, we study how network traffic is affected as we increase the number of indexed queries resulting in the execution of a larger distributed NFA. The results are shown in Figure 4.19. We can see that network traffic scales linearly with the number of queries. As we index more queries in FoXtrot, the part of the distributed NFA that we traverse during filtering is larger and as a result more messages travel through the network.

We also examine how the number of predicates per query affects network traffic during filtering. In this case we increase the number of predicates included in each query. Figure 4.19(b) shows the total amount of network traffic generated during filtering against the corresponding sets of queries with 1, 2, and 3 predicates. We present two cases when $5 * 10^5$ and 10^6 queries are indexed respectively. In both cases, the query set which contains more predicates is more selective and this results in traversing a smaller part of the NFA during filtering. So, network traffic is significantly decreased as the number of predicates per query increases as we can see in Figure 4.19(b).

We continue by demonstrating how parameter l affects the number of messages that are generated during filtering. The results are shown in Figure 4.20, where we measure network traffic as we increase l. We repeat our experiment for the two cases where the average document depth is 5 and 10 respectively. As Figure 4.20 depicts, increasing the value of parameter l results in decreasing the generated amount of network traffic. This is explained as follows. When l is increased, each peer is able to perform execution on a larger path

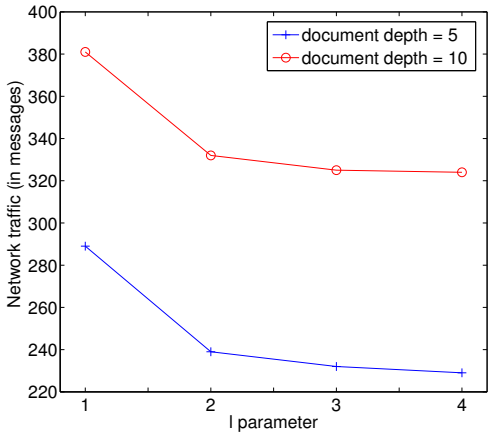
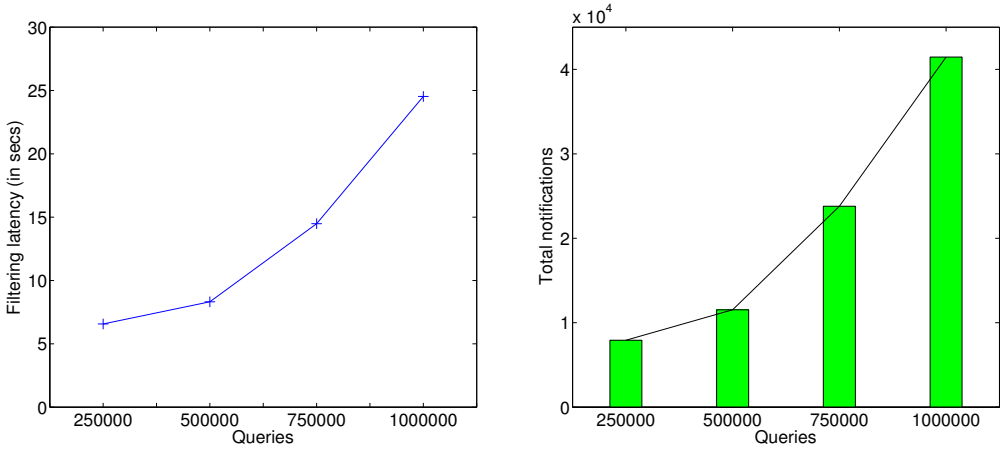


Figure 4.20: Increasing parameter l



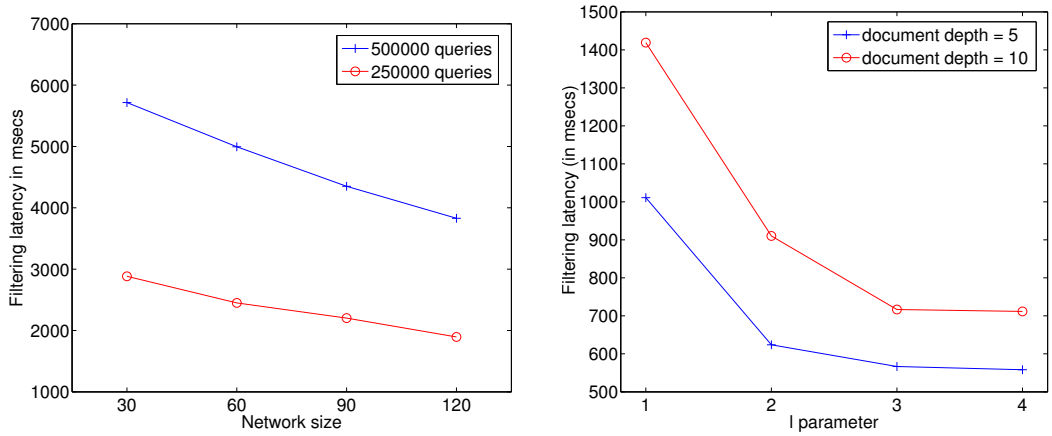
(a) Increasing indexed queries (b) Notifications

Figure 4.21: Filtering latency and notifications

of the distributed NFA. However we can see that as l is increased more, the corresponding decrease in the generated traffic is smaller.

Filtering latency and throughput Apart from network traffic, we are mainly concerned with the filtering latency of the XML documents that arrive in FoXtrot. Recall that for a set of XML documents D, we measure filtering latency as the amount of time spent until all notifications are disseminated to the interested subscribers for the queries satisfied by the documents of D. As a result, filtering latency strongly depends on the number of notifications that are generated during filtering.

We begin by studying how filtering latency is affected as we increase the total number of indexed queries. The results are shown in Figure 4.21(a). As the graph shows, filtering



(a) Increasing network size

(b) Increasing parameter l

Figure 4.22: Filtering latency

latency is increased when the number of queries indexed in FoXtrot is increased. More specifically, the time spent for filtering and delivering the notifications is proportional to the number of the queries matched in terms of the generated notifications. The number of these matches in each case is depicted in Figure 4.21(b). For example, $4 * 10^4$ notifications are generated when matching against 10^6 queries (selectivity of 4%). In terms of throughput, when 10^6 queries are indexed in FoXtrot, after publishing 100 XML documents, FoXtrot generates and disseminates about 1600 notifications per second.

We continue by demonstrating the scalability of FoXtrot during filtering as we increase the size of the network. We repeat our experiment for networks consisting of 30, 60, 90, and 120 peers accordingly. The results are shown in Figure 4.22(a) where two cases are depicted, when $2.5 * 10^5$ and $5 * 10^5$ queries are indexed respectively in the system. As the results clearly indicate when the size of the network increases, the time for filtering is significantly decreased. For example, when network size is increased from 30 peers to 120 peers and $5*10^5$ queries are indexed in the system, filtering latency is decreased from 6 to less than 4 seconds.

We also study how we can improve filtering performance by increasing the value of parameter l. The results are shown in Figure 4.22(a). As expected, filtering latency is significantly decreased as we increase l and this is mainly due to the smaller amount of network traffic that is generated (studied earlier and depicted in Figure 4.20). Also we observe that the margin for improvement is larger when the document set being filtered includes XML documents of a higher depth.

4.6.3 Discussion

Let us now summarize the results from our experimental evaluation. First, with respect to query indexing, FoXtrot is highly efficient reaching a throughput of almost 1000 queries per second for a network of 112 peers deployed using the cluster machines. Even though we cannot directly compare FoXtrot to other systems evaluated under different conditions we report on the performance of XNet [22] which is a closely related system for distributed XML filtering. XNet is evaluated using an overlay consisting of 22 peers from the PlanetLab network. A total of 10^5 queries are indexed in XNet which exhibits an indexing throughput of almost 19 single-element (i.e., query depth is 1) queries per second for each peer.

We also studied the performance of FoXtrot during filtering by publishing a burst of 100 XML documents and FoXtrot exhibited a high filtering throughput generating and delivering about 1500 notifications per second. Moreover, we demonstrated how scalable FoXtrot is since increasing network size results in improving performance and decreasing latencies. For load balancing, we employed two simple yet effective replication methods for distributing the load among the FoXtrot peers. We exhibited that using our dynamic replication method we can evenly distribute the filtering load while incurring a small storage overhead to the peers. We also illustrated how parameter l affects the performance of FoXtrot and showed experimentally that increasing l can help us decrease network traffic and improve filtering latency especially for the case of deep XML documents. In general, depending on the specific properties of the distributed NFA and the size of the network, tuning parameter l can lead to an improved performance.

The majority of these experiments were conducted on two different environments, namely the PlanetLab network and a shared cluster. PlanetLab network represented the real-world conditions of the Internet and for this purpose we deployed FoXtrot on 396 nodes. The latency observed in PlanetLab, either during indexing or filtering, was always one order of magnitude higher than the one observed in the cluster. We also experienced an increased variation in the measurements of Planetlab among our different runs. This was mainly due to the existence of slow nodes in PlanetLab. [86] have studied this problem in the context of a simple DHT service deployed using PlanetLab machines and propose several methods for overcoming its effect.

4.7 Summary

In this chapter, we first described in detail how we construct, maintain and execute the distributed NFA FoXtrot. Then, we described methods for load balancing and techniques for improving the fault tolerance of the system. We concluded this chapter with an extensive

experimental evaluation. While our approach and other similar approaches that employ automata or similar indices have been used with success for structural matching, little attention has been paid to predicate evaluation or else value matching. The next chapter describes and evaluates different methods for combining structural and value matching in FoXtrot.

Chapter 5

Value matching

While our approach and other similar approaches that employ automata or similar indices have been used with success for representing a set of queries and identifying XML documents that structurally match XPath queries, little attention has been paid to the evaluation of value-based predicates, especially in distributed settings. However, there are many cases where predicate evaluation can be costly and attention should be paid for providing an efficient solution. To the best of our knowledge, our work is the first to study combined structural and value matching of queries in a distributed setting aiming to distribute both tasks. In this chapter, we describe different methods for distributing the task of value matching in combination with our structural matching algorithms described earlier.

5.1 Overview

First, we give a high-level overview of the problem and the different methods one can use to combine structural and value XML filtering in general and then we describe in detail how we employ these methods in our distributed setting. Consider for example query q: `/dblp/phdthesis[author ="`*Iris Miliaraki*`"]`, which selects the Ph.D. thesis of author Iris Miliaraki. Filtering incoming XML data against q requires to check whether the data structurally match the query and also whether the value-based predicates of q are satisfied. This can become an important problem depending on the selectivity of the structural and value-based predicates. So the number of queries which are only structurally matched might be large. We will refer to queries only structurally matched as false positives. Our goal is to design a system that scales with respect to both the number of the queries indexed and the number of the predicates included in the queries. Let us now discuss the alternative methods we can employ for achieving this in the distributed environment of FoXtrot.

Following a widely used strategy from relational query optimization, where selections are

applied as early as possible, we can check the value-based predicates before proceeding with the structural matching following a *bottom-up approach*. Such an approach evaluates XML documents in a *bottom-up way* since in a tree representation element values are placed in the leaves of the tree. To support such a method, we first want to discover queries that contain specific predicates. This is accomplished by mapping queries to peers using their predicates. After discovering a set of queries that contain a specific predicate, we will then perform structural matching. This indexing algorithm resembles works presented for information filtering (IF) on top of DHTs including work of Tryfonopoulos et al. [100], where queries are expressed using a simple attribute-value data model and attribute values are used in this case to map queries to peer identifiers. As expected, we call this method *bottom-up evaluation*.

Alternatively, we can evaluate the predicates after performing structural matching. Such a technique operates in a *top-down fashion* processing incoming XML documents from the root towards their leaves. In this case, contrast to the previous approach, we can employ the distributed NFA to identify the subset of queries that structurally match incoming XML documents, and then evaluate the predicates of the corresponding subset of queries. Hence, this method evaluates predicates after the execution of the NFA. Since in FoXtrot structural matching is performed in parallel by multiple peers, each of these peers identifies a different subset of structurally-matched queries. Whenever a peer identifies such a set, it is also responsible for the predicate evaluation. We refer to this method as *top-down evaluation*.

Furthermore, considering that XPath queries consist of distinct steps and each step may be associated with one or more value-based predicates, we can perform at each step of the NFA execution structural matching concurrently with predicate evaluation. In this case we evaluate predicates during the NFA execution in *step-wise* manner. We refer to this method as *step-by-step evaluation*.

Finally, since in our case the XPath queries are indexed using an NFA, we could perform predicate evaluation directly with the automaton by adding extra transitions for the predicates. An expected drawback of such a method comes from the fact that the elements in a set of XPath queries represent a rather small set since they are constrained by the schema, while the values of the predicates may form a really large set. This could result in a huge increase of the NFA states and most importantly destroy the sharing of path expressions for which the NFA was selected to begin with. For this reason, we have not studied this method any further.

In the following, we describe how we implement the above methods, namely *bottom-up*, *top-down* and *step-by-step evaluation* for offering XML filtering functionality on top of Pastry DHT. In the case of the bottom-up evaluation method, since predicate evaluation precedes structural matching, a different query indexing algorithm is used to distribute queries in

the network. Instead of using the path structure of the queries to traverse the NFA, we index them based on their predicates. Top-down evaluation and step-by-step are designed assuming that queries are indexed using the distributed NFA described earlier (see Chapter 4). In addition, we propose a method called *top-down with pruning* which improves on top-down by applying some early checks during structural matching.

5.2 Prerequisites

In this section, we describe our query model focusing on the types of predicates allowed, our data representation model and also define some terms used in this description.

5.2.1 Revisiting our data and query model

As described in detail in our data model we allow path queries containing attribute and textual predicates. Without loss of generality we describe our value matching techniques assuming that each query contains *at least one predicate*. Also, based on the subset of XPath we allow, we consider only conjunctions of predicates in the queries.

With respect to our data model, recall that for structural matching to take place we enrich the XML documents that arrive for filtering with a positional representation. Such a representation helps us to efficiently check structural relationships between elements. For the purpose of predicate evaluation we also generate and attach with this representation a set of *candidate predicates*. We call these predicates candidates because they correspond to the query predicates that can be satisfied by the incoming XML documents. Such a representation (i.e., positional information and candidate predicates) requires a single pass over the input XML document.

Depending on the predicates we allow in the subset of XPath supported in our setting, we can distinguish among different ways for constructing the candidate predicates. We decide to focus only on equality predicates and consider only this type of predicates in the rest of this thesis. So, each candidate predicate is an equality predicate constructed using either an element name, an attribute and its value (*attribute predicates*) or the element name and its text value (*textual predicates*), as found in the XML data fragment.

The publisher peer is responsible for enriching the parsing events and producing the set of the candidate predicates. The enriched parsing events along with the candidate predicate set are forwarded with each filtering request. Note that we parse the document a priori to generate the above structures before proceeding with filtering. However, we do not consider this additional parsing operation to cause significant load to the system since it is typical for XML dissemination systems to deal with relatively small documents. As mentioned in

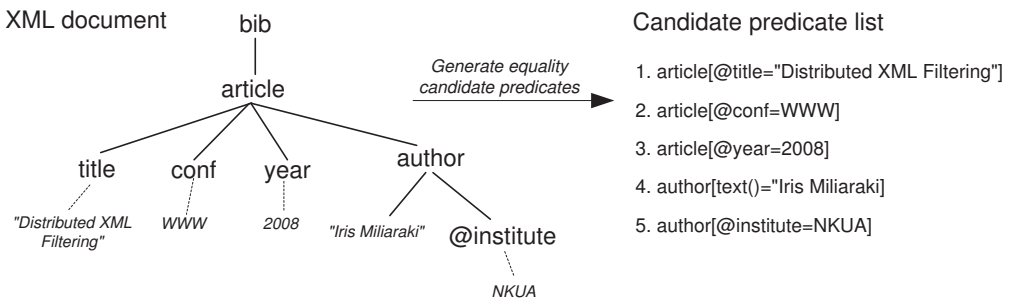


Figure 5.1: Constructing candidate equality predicates

the study of Barbosa et al. [11], the average size of an XML document in the Web is only 4 KB, while the maximum size can reach 500 KB.

5.2.2 Terminology

In the following we give some definitions that will be used in the rest of this chapter.

Definition 5.2.1. *An* NFA path *is a sequence of NFA states $st_0, st_1, \ldots, st_n$ such that for every pair states st_i,st_{i+1} there exists an input symbol w where $\delta(w, st_i) = st_{i+1}$ (i.e., there exists a transition from st_i to st_{i+1}).*

Definition 5.2.2. *An* NFA accepting path *of a query q is an NFA path $st_0, st_1, \ldots, st_n$ where st_0 is the start state of the NFA and st_n is the accepting state of q.*

Example 5.2.1 (Generating candidate predicates)**.** *An example of how we construct the set of candidate predicates from an XML document is depicted in Figure 5.1. The XML document refers to an article and XML element* `article` *has an attribute* `title` *with value "Distributed XML filtering". The corresponding candidate predicate has the form* `article`[`@title="`Distributed XML Filtering`"`]. *If we want to also support other types of predicates, like range predicates, then we should construct our candidate predicates in a different way.*

5.3 Methods

We now describe in detail our methods, namely *bottom-up*, *top-down* and *step-by-step evaluation* for offering distributed combined structural and value matching in FoXtrot.

5.3.1 Bottom-up evaluation

Inspired by the heuristic from relational query processing to push selections as early as possible, we describe a method that operates in a bottom-up fashion. This method indexes

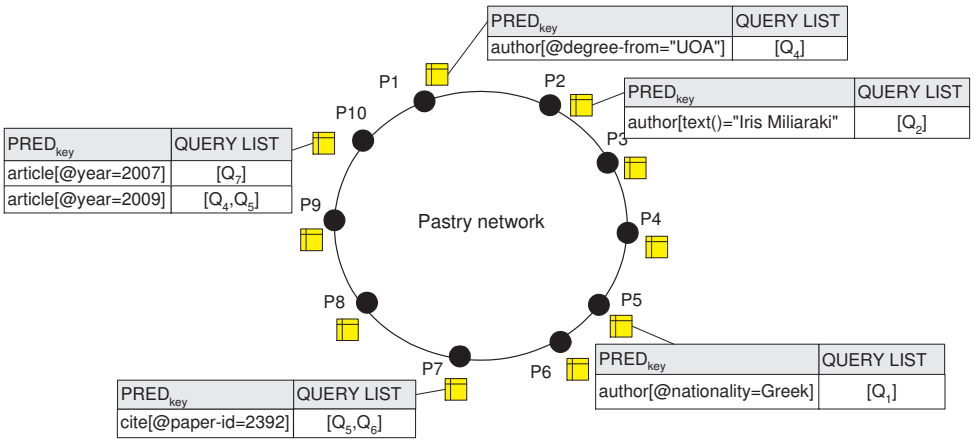


Figure 5.2: Query indexing in bottom-up approach

queries in the network based on their predicates and performs filtering by first checking the value-based predicates and then proceeding with the structural matching. Contrast to the other methods presented here, which operate in a complementary way to our structural matching techniques, bottom-up evaluation requires a different indexing approach.

Indexing queries We assign each query to a network peer using its predicates. So, for each distinct predicate p included in a query set, there is a single responsible peer which is responsible for any query that contains this predicate. Each predicate is uniquely identified by a key formed by its string representation to determine the responsible peer. Formally, since we employ the Pastry DHT as the underlying network, the *responsible peer* for predicate with key k is the peer whose identifier is numerically closest to $Hash(k)$, where $Hash()$ is the DHT hash function. Locally, each peer maintains a hash index mapping predicates to the list of queries that contains them for fast retrieval. Each query is indexed multiple times in the network, depending on the number of distinct predicates it contains.

Example 5.3.1 (Query indexing in bottom-up approach)**.** *We demonstrate with an example how queries are assigned to network peers to enable bottom-up evaluation. The set of queries indexed is shown in Figure 5.3 along with the peers responsible for each different predicate. For the purposes of this example, we associate a number with each distinct predicate. Figure 5.2 illustrates the local hash indices kept by the peers for indexing the queries.*

Filtering data Whenever an XML document arrives for filtering, we construct the set of candidate predicates. For each candidate predicate, we create and send a filtering request to the peer responsible for this predicate. The peer probes its local index, retrieves the queries that contain the specific predicate and then performs *locally* structural matching for

Queries

Q_1: /bib/phdthesis[@published=2005]/author[@nationality=greek] (1, 2)

Q_2: /bib/*/author[text()="John Smith"] (3)

Q_3: /bib/article[@conf=debs] (4)

Q_4: /bib/article[@year=2009]/author[@degree-from="UOA"] (5, 6)

Q_5: /bib/article[@year=2009]/cite[@paper-id=2392] (5, 7)

Q_6: /bib/article/cite[@paper-id=2392] (7)

Q_7: //article[@year=2007] (8)

Predicate assignments to peers

Predicate key	Responsible peer
1	P8
2	P4
3	P2
4	P3
5	P10
6	P1
7	P7
8	P10

Figure 5.3: Queries and predicate assignments to peers

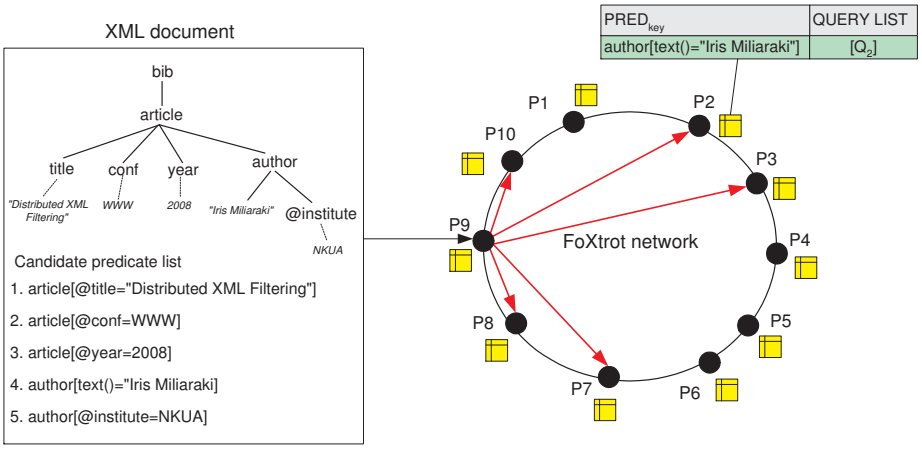


Figure 5.4: XML filtering in bottom-up approach

these queries. Structural matching can be performed using any centralized engine for this purpose. We prefer to use an NFA like in the case of YFilter system [30]. So, if a query is also structurally matched against the incoming XML document, a notification is sent to the interested subscriber. Since we construct several candidate predicates from each XML document, filtering is performed in parallel by the corresponding peers which are responsible for these predicates. Specifically, each peer performs structural matching for the subset of queries that contain the predicates that led to that peer.

Example 5.3.2 (Filtering in bottom-up approach)**.** *Figure 5.4 demonstrates an example of how filtering takes place when an XML document arrives. Suppose that the XML document depicted in Figure 5.1 arrives for filtering at peer P9. For each candidate predicate constructed, the publisher peer issues a different filtering request and forwards it to the peer responsible for this predicate. Then, the peer retrieves from its local index the set of queries that contain this predicate and performs structural matching for this set. In this example, a match is found by peer P2 for predicate* `author[text()="Iris Miliaraki"]` *which is*

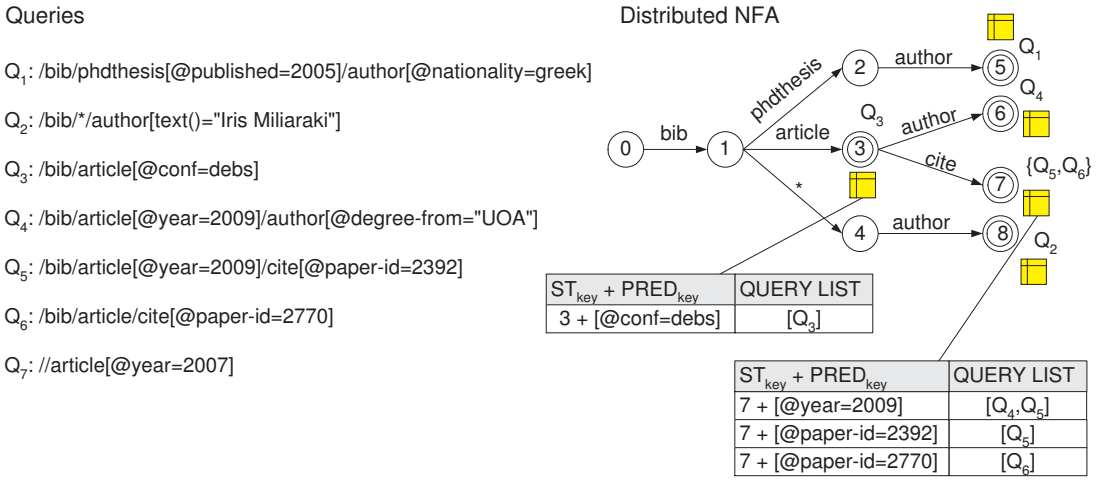


Figure 5.5: Query indexing in top-down evaluation

contained in query Q2. If the query is also structurally matched and the rest of its predicates are satisfied, a notification is sent to the interested subscriber.

5.3.2 Top-down evaluation

Instead of first evaluating predicates, this approach begins by performing structural matching. In this case, we use the distributed NFA to identify the subset of queries that structurally match incoming XML documents, and then we evaluate the predicates of these queries. In other words, this method evaluates predicates after the execution of the NFA. Since structural matching is performed in parallel by multiple peers, each of these peers identifies a different subset of structurally-matched queries and as a result the task of predicate evaluation is also distributed among the network peers. Whenever a peer identifies such a set, it is also responsible for the predicate evaluation.

Indexing queries Query indexing is performed using the distributed NFA. When we reach an accepting state then predicate evaluation is performed for the set of queries associated with that state. We avoid evaluating each predicate separately by utilizing an index structure. Since we deal with equality predicates, it is sufficient to construct a hash index mapping predicates to the list of queries which contain them. For each accepting state, we include in the hash index all the predicates of the corresponding queries. Since each peer can store more than one accepting state, each entry in the index maps $\{ST_{key}, PRED_{key}\}$ pairs, where ST_{key} is the key of the accepting state and $PRED_{key}$ is the predicate key, to a list of queries. We discuss later how we can extend our method to support other types of predicates like range ones.

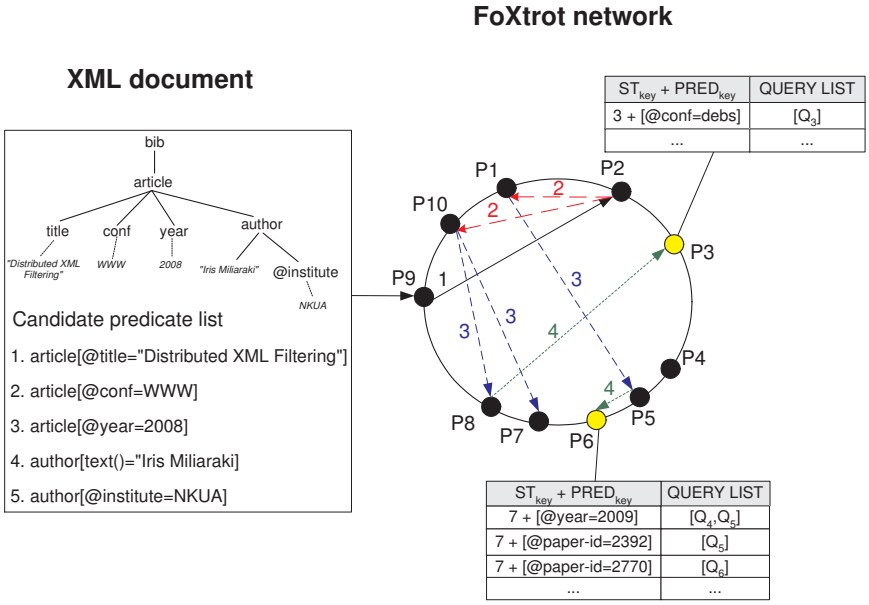


Figure 5.6: XML filtering in top-down evaluation

Example 5.3.3 (Query indexing in top-down approach)**.** *We illustrate the above using an example shown in Figure 5.5. In the top-down evaluation we associate each accepting state with hash indexes (i.e., states* $3, 5, 6, 7$ *and* 8 *denoted by two concentric circles). The hash index of state* 3 *contains a single entry for* Q_3 *and the index of state* 7 *contains three entries, one for each distinct predicate contained in* Q_5 *and* Q_6*.*

Filtering data Filtering is performed by executing the distributed NFA until we reach an accepting state. When a peer reaches an accepting state, it needs to further evaluate the queries associated with that state with respect to their value-based predicates. Instead of sequentially checking all the queries that have been structurally matched, we use the candidate predicates to probe the hash index. If a query is matched by incoming data it will then be returned as a match by the index. If all the predicates of the query are satisfied we notify its subscriber.

Example 5.3.4 (XML filtering in top-down approach)**.** *Let us illustrate with an example how we filter an XML document using top-down approach. Suppose that the XML document depicted in Figure 5.6 arrives for filtering at peer* $P9$*. The execution of the NFA begins by contacting peer* $P2$ *which is responsible for the start state. Execution proceeds as usual until we reach an accepting state. In the example shown in Figure 5.6, states* 3 *and* 7 *are accepting states. Peer* $P6$ *reaches accepting state* 7 *and performs predicate evaluation for the queries associated with state* 7*. As we explained earlier, to avoid a sequential check of these queries which may constitute a large set, peer* $P6$ *uses its local index and retrieves only the*

queries that contain at least one of the candidate predicates. Peer P3 which also reaches an accepting state operates in like fashion. If a complete match is found, subscribers of satisfied queries are notified.

5.3.3 Top-down evaluation with pruning

We investigate an improvement on top-down evaluation for overcoming the problem of spending too much effort structurally matching queries with predicates that are not satisfied by the incoming XML documents. We already discussed that it can be very expensive, in terms of NFA states, to represent directly with transitions the value-based predicates. Instead, we propose to use a compact summary of predicate information to stop the execution of the NFA (i.e., *prune* an execution path) whenever we can deduce that no match can be found if the execution continues. We refer to this method as *top-down evaluation with pruning*. The basic idea of this approach is the following. Recall that the NFA is a tree structure distributed among the network peers and during structural matching we traverse it by visiting the peers responsible for its different NFA fragments. Each peer can be responsible for storing many fragments of this NFA. At each step of the execution, we can consider that a part of the NFA has been *revealed* while the rest part is not. So, we propose to use a data structure for representing these NFA fragments with respect to the predicates they contain. Given that we want to support a large set of queries, we prefer to use for this purpose a probabilistic structure and specifically *Bloom filters* [12]. The main idea is that we construct these filters incrementally during indexing, while at the time of filtering we consult them and decide whether we should continue or terminate execution earlier. In the following, we first describe in detail the commonly used data structure of Bloom filters [12] and then describe our method.

Bloom filters Bloom filters were proposed by Burton Bloom [12] for the probabilistic representation of a set to support membership queries. A Bloom filter is a bit-vector of length m used to represent a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements. Initially all bits are set to 0. Then, using k independent hash functions $h_1, h_2, \ldots, h_k$ with range 1 to m, each element $x \in S$ sets to 1 the bits of positions $h_i(x)$ for $1 \leq i \leq k$. Each bit can be set to 1 many times, but only the first operation has an effect. Then, to check whether an item y is in S, the bits at positions $h_1(y), h_2(y), \ldots, h_k(y)$ are checked. In case any of them is 0 then y is not a member of S, else we assume y is in S. There is however a probability that this is a false positive and it has been shown that this probability is equal to $(1 - e^{kn/m})^k$.

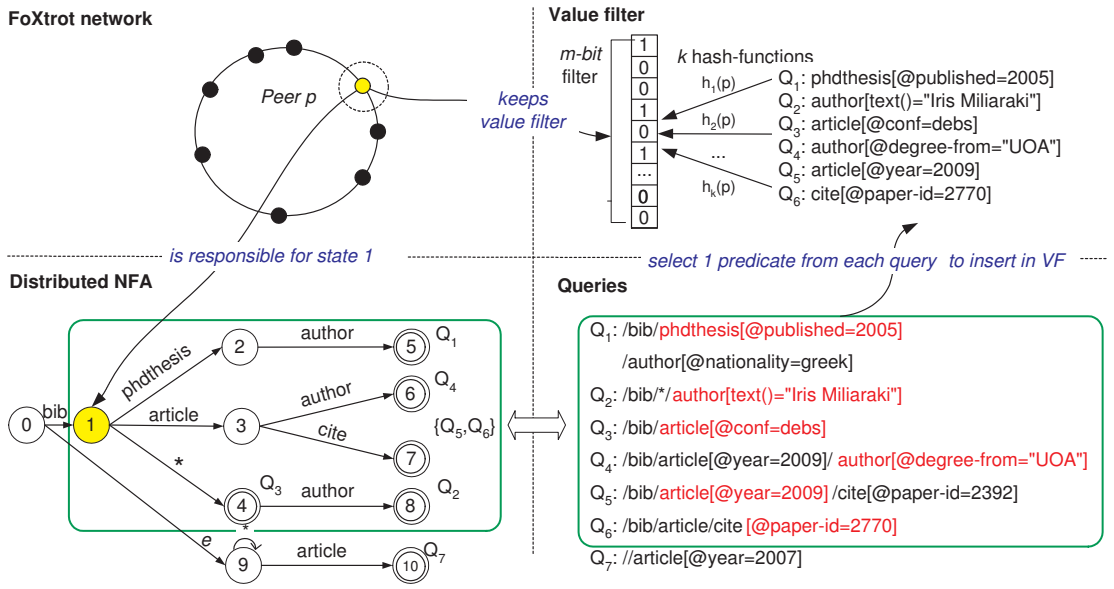


Figure 5.7: Query indexing in top-down evaluation with pruning

Indexing queries The main idea of this method is to use Bloom filters for summarizing the query predicates indexed in a specific NFA fragment. For easier maintenance each peer keeps a single Bloom filter to summarize a set of value-based predicates. We call such filters *value filters* (VF). Each predicate represented by a value filter is associated with the respective NFA state. Query indexing is performed as described earlier but instead of only updating the distributed NFA, we also update the corresponding value filters using the query predicates. Since we assume only conjunctions of predicates in queries, if at least one of the query predicates is not satisfied, then the query cannot be matched. Consider a peer p and a state st for which p is responsible for. Based on the former observation, for each query q whose NFA accepting path contains st, we insert one predicate of q in the VF of p. In other words, only one predicate of each query is required in the value filter. Each attribute predicate of the form $element[@attr = value]$ is inserted as a whole in the VF using its string representation $element + attr + value$ concatenated with the state identifier. Likewise, textual predicates of the form $element[text() = value]$ are inserted as a whole in the VF using their string representation $element + text() + value$ together with the state identifier. We insert predicates in the filters during query indexing. Since we need to traverse the NFA accepting path of each query in order to index it, it is guaranteed that all relevant VFs will be updated. For a query which contains more than one predicates, we need to decide which one will be inserted in the value filter. We can either do this in a random or a more sophisticated way. We discuss these alternatives later on in Section 5.4. Alternatively, instead of only using the name of the element for inserting it in the VF, we can use the path

of elements ending with the corresponding element associated with the predicate. However, this causes overhead in the process of constructing and using value filters which we prefer to avoid. In addition, this can be considered a schema-dependent decision since recursive XML schemas allow to repeat the same element in a query while others do not.

Example 5.3.5 (Query indexing in top-down with pruning approach)**.** *We illustrate the above using an example shown in Figure 5.7. We consider queries* $Q_1, Q_2, Q_3, Q_4, Q_5, Q_6$ *and* Q_7 *and the corresponding NFA. This NFA is distributed among the network peers. In our example, peer p is responsible for state* 1*. Since state* 1 *belongs to the NFA accepting paths of queries* Q_1, Q_2, Q_3, Q_4, Q_5 *and* Q_6*, one predicate of each query is inserted in the value filter of p for state* 1*. Note that we do not insert a predicate of* Q_7 *since* Q_7 *is not indexed in the corresponding NFA fragment i.e., its NFA accepting path does not include state* 1*.*

Filtering data In this method, we require from each peer participating in the execution process to perform an additional step before expanding an NFA state. During this step, peer checks whether its VF matches any of the candidate predicates. In case no candidate predicate can be matched (*miss*), execution is terminated instantly, while if *at least one* of the candidate predicates is found in the filter (*match*), the peer continues execution. In the worst case where no execution path is pruned, this method works exactly like the top-down evaluation with the small overhead of checking the bloom filters during the execution. As described previously, when we reach an accepting state each peer keeps a hash index for performing predicate evaluation in the same way as in top-down method. Note that we do not explicitly check query predicates, but at each step of the execution, the value filter only gives us enough information regarding whether to continue the execution or not. We have no further information regarding which queries contain the candidate predicates that cause execution to continue since bloom filters only answer membership queries.

Example 5.3.6. *We demonstrate how we use value filters to prune NFA execution using an example shown in Figure 5.8. Again, we consider the same set of queries and NFA distributed among the network peers (see Figure 5.7). In our example, peer* p_1 *is responsible for state* 0*, while peer* p_2 *is responsible for state* 1*. Suppose a peer publishes the XML document. This peer is responsible for generating candidate predicates from the document and initiating filtering. Peer* p_1 *which is responsible for the start state receives the filtering request and begins filtering. Before proceeding with the expansion of state* 0*, it checks whether any of the candidate predicates is included in its local value filter. In this case, a match is returned for predicate* $article[@year = 2007]$*, included in query* Q_7*. So, state* 0 *is expanded causing states* 1 *and* 9 *to become active. Peer* p_2 *continues execution from state* 1*. In this case, when* p_2

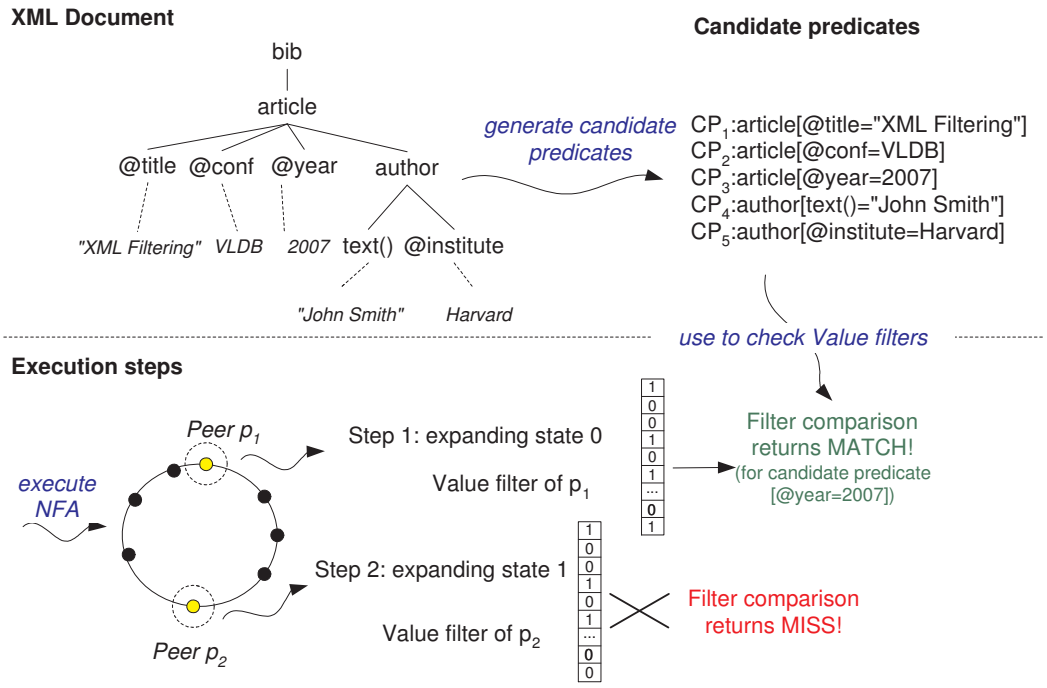


Figure 5.8: XML filtering in top-down evaluation with pruning

checks its local value filter, it founds no match and this execution path is pruned and state 1 *is not expanded. To avoid complicating this example, we do not depict the execution path from state* 9 *in the figure.*

5.3.4 Step-by-step evaluation

We conclude our description about value matching with our last method based on the concept of checking predicates concurrently with performing structural matching. This method is called *step-by-step* since it performs filtering in a step-wise manner and evaluates predicates during the execution of the NFA.

Indexing queries Similarly to the other methods, queries are indexed using the distributed NFA. During query indexing, each peer organizes all the predicates included in its local queries using an index. Each predicate is associated with the relevant NFA state it refers to and the local index maps predicates to the list of queries which contain them. Since a query may not contain predicates at all steps, we also map a generic `true` predicate to queries with no predicates at this step. Contrast to top-down method where we only keep indices for the accepting states, in step-by-step, indices contain entries for all NFA states. Note that each peer keeps locally a single index.

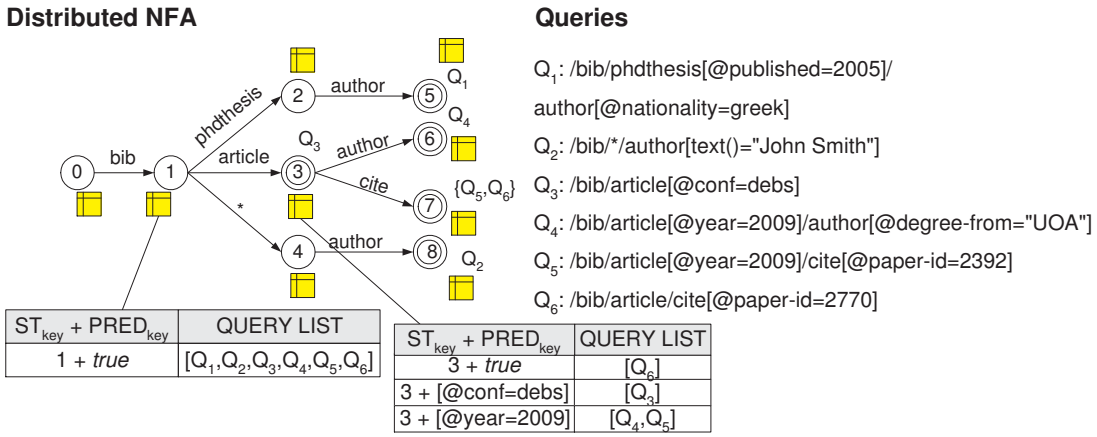


Figure 5.9: Query indexing in step-by-step

Example 5.3.7. *Figure 5.9 illustrates the above with an example. We consider again the same set of queries. We also depict the NFA which is distributed among the network peers. Each NFA state is associated with a number of predicates. For each distinct predicate, there is an entry in the corresponding index mapping it to the queries that contain it. So, for example for state* 1 *we create an index entry mapping a generic* true *predicate to queries* Q_1, Q_2, Q_3, Q_4, Q_5 *and* Q_6 *since no predicates are associated with this state. Instead, for state* 3 *we keep* 3 *different entries,* 1 *for the* true *predicate and* 2 *for the distinct predicates associated with that state. Note that if the same peer* p *was responsible for states* 1 *and* 3*, then all entries would be part of a single index kept locally by* p*. However, for the sake of simplicity we do not depict the corresponding peers in this example.*

Filtering data Recall that each filtering request carries a document representation and the candidate predicates produced by the document. In addition, each filtering request also includes the queries which have been partially matched with respect to both their structure and predicates until the current execution step. To filter incoming XML data, each peer participating in the filtering process uses the candidate predicates to probe its local index.

Let us explain how this works in detail. After peer p expands a state st during execution, it uses the candidate predicates to probe its hash index. Let $CurrQ$ and $PrevQ$ be the set of queries returned by the hash index and the set of previously satisfied queries respectively. We will denote as $NextQ$, the set of satisfied queries after expanding st, i.e., $NextQ = PrevQ \cap CurrQ$. In case state st is an accepting state for a query $q \in NextQ$ (both structure and predicates have been satisfied), then p notifies the subscriber of q and removes q from $NextQ$. Then, execution continues and p forwards a new filtering request that includes $NextQ$. At the current state of our work, we only support equality predicates and for this reason we use a *hash index*. We plan to support range predicates using appropriate indexes

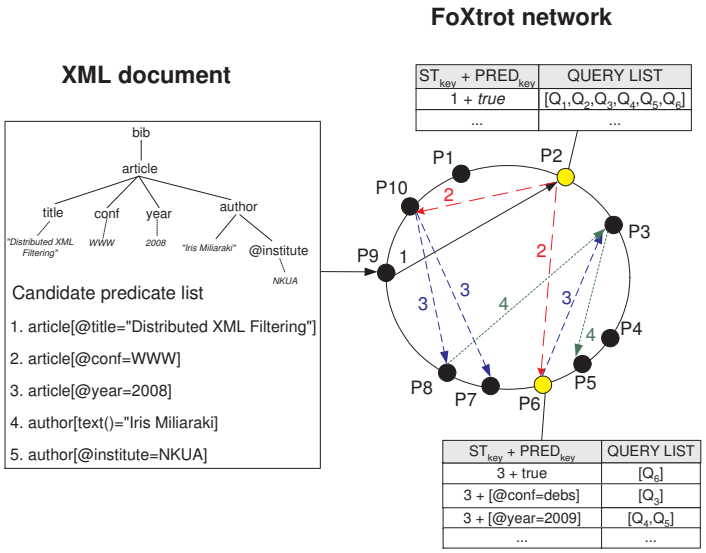


Figure 5.10: XML filtering in step-by-step

like B^+-trees.

We illustrate the above using an example shown in Figure 5.10. We construct an NFA from a set of six queries. For the sake of simplicity, we do not show how this NFA is distributed among network peers. For each NFA state, we keep a different hash index. In Figure 5.10, we show the contents of the hash indexes associated with states 1 and 3 respectively. The index at state 1 contains only one entry mapping the *true* predicate to $\{Q_1, Q_2, Q_3, Q_4, Q_5, Q_6\}$, since no query contains a predicate at that step. While, the index at state 3 contains three entries, one for the *true* predicate and two for the predicates contained in queries Q_3,Q_4 and Q_5. Q_4 and Q_5 contain the same predicate, so one entry is added to the hash index. Execution ends when $NextQ$ becomes empty.

5.4 Online selectivity estimation

In this section, we define the selectivity of value-based predicates in our context, describe techniques for estimating their selectivity and explain how we can exploit these statistics for optimizing our value matching method of top-down evaluation with pruning. Even though the problem of selectivity estimation for XML path expressions has been studied extensively in the past, little attention has been paid in estimating the selectivity of the value-based predicates included in the queries.

5.4.1 Overview

Our value-matching method employs bloom filters to summarize predicate information and prune execution as early as possible. We achieve this by deducing that no match can be found if we continue execution. For each query that we want to summarize its predicates in the corresponding filter, we select and insert only one of its predicates. Recall that we support linear path queries which can contain at each query step conjunctions of predicates. This selection can be made *randomly* or ideally we can select the predicate which can lead to better pruning, i.e. the least selective predicate.

Example 5.4.1 (Predicate selectivity)**.** *Consider query Q:* `/dblp/article[@year = 2009]/author[text() = "Michael Smith"]`, *which selects all articles of author Michael Smith which were published in 2009. Intuitively, for this example, one can expect that there are many more articles published in 2009 than the total articles of author Michael Smith. As a result, we would like to insert in the filter the predicate on the author.*

In general, we want to select and insert the *most selective predicate* of each query in the respective filters. In the following, we first define predicate selectivity in our context, describe how we estimate this selectivities in our distributing setting, and explain how these statistics are used by our top-down method for efficient pruning.

5.4.2 Definitions

As described in this chapter, we support two types of predicates in XPath queries, namely attribute and textual predicates on XML elements. In the following we define predicate selectivity for these two types of predicates.

Definition 5.4.1. *We define the* selectivity of a textual predicate $[text() = v]$ *on element e for an XML document collection D as the fraction of elements e, reachable by any path, with value v in D. The corresponding formula is the following:*

$$sel(e[text() = v]) = \frac{total\ occurrences\ of\ e\ equals\ v\ in\ D}{total\ occurrences\ of\ e\ in\ D} \tag{5.1}$$

Definition 5.4.2. *We define the* selectivity of an attribute predicate $[@attr = v]$ *for attribute attr on element e for an XML document collection D as the fraction of elements e, reachable by any path, that satisfy the predicate in D. The corresponding formula is the following:*

$$sel(e[@attr = v]) = \frac{total\ occurrences\ of\ attr\ equals\ v\ in\ D}{total\ occurrences\ of\ e\ in\ D} \tag{5.2}$$

We also consider that any predicate on an element explicitly defined is more selective than a predicate on a wildcard element.

5.4.3 Distributed sampling

In an XML dissemination system like the one described in this thesis, a large volume of XML data is expected to arrive for filtering. As a result, it is not feasible to store the entire set of XML data that have been processed for computing the exact predicate selectivities. For this reason, we use sampling and keep information only about a fixed-size random subset of the XML data collection.

Formally, we want to obtain a random sample of size n from a set of size N, where N is not known beforehand. A well-known and broadly used algorithm for random sampling is the *reservoir sampling algorithm* proposed by Vitter et al. [104]. For obtaining a random sample of size n, the algorithm works as follows. The first n items are inserted in the reservoir. Then, a random number of items is skipped and the next item replaces a random item from the reservoir. Again, a random number of items is skipped and so on. This way the reservoir always contains a random sample of n items.

Let us recall at this point what kind of selectivities we want to extract from the document sample. We keep the occurrences of each element along with the occurrences of its attributes and their values. Each peer keeps its own independent reservoir of n items, in our case XML documents. This reservoir is created from the documents that arrive at this peer. This random sample is kept at the document-level, so each item in the reservoir is actually an XML document that has previously arrived for filtering. Whenever a document arrives at a peer, it runs the reservoir sampling algorithm to decide whether or not to add this document to its local reservoir.

An alternative method for improving our estimates is instead of having each peer keeping its own independent sample, use a distributed sampling algorithm. Pitoura and Triantafillou [79] describe several distributed sampling algorithms on top of DHTs for estimating peer loads. These algorithms can be adapted to obtain better estimates of predicate selectivities by combining the estimates computed by different peers. For instance using the random walking algorithm described in this work [79], each peer can collect the estimated selectivities from each neighbors and compute better estimates.

Another factor that can affect the quality of our estimates is that we perform sampling at the XML document level. As stressed by Chand et al. [23], where the authors estimate tree pattern selectivities, such a coarse-grain document-level sampling can result in poor estimates. By keeping more fine-grained statistics at the level of distinct elements, the quality of the statistics can be improved. Such a technique is described in the work of Chand et

al. [23] where the authors maintain a sample of distinct elements over an XML document stream to collect tree pattern selectivities. Even though the authors do not consider predicate selectivities, the core ideas described for path selectivities can be also applied for the estimation of predicate selectivities.

Since our focus is not on estimating the selectivities but demonstrating that exploiting such statistics instead of making a random choice can lead to an improvement of our method, we do not consider them any further in the context of this thesis.

5.4.4 Using statistics in predicate evaluation

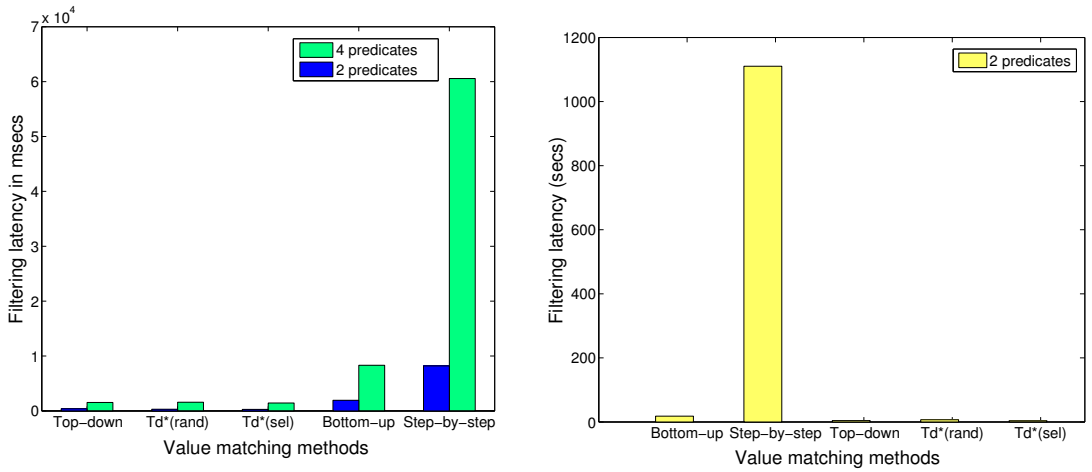
We have described what kind of selectivity statistics we keep and how we compute them. Let us now describe how these statistics are exploited by the value-matching method to select which predicates will be inserted in the value filters.

Each time a query arrives at a peer p, before starting indexing the query, p computes the selectivities of its predicates by consulting its local reservoir. Then, p marks the most selective predicate of the query and initiates indexing. This process is performed only by the subscriber peer. Recall that we update value filters for a specific query only once when it arrives for indexing, so the most selective predicate is chosen using the selectivity statistics collected until that point of time. Alternatively, we could also update filters at a later time because XML data may follow a different distribution and the selectivities may change. However, this would require to be able to also remove entries from the Bloom filters. In this case, one could use Counting Bloom filters [32].

5.5 Experimental evaluation

In Chapter 4, we evaluated the performance of FoXtrot in the absence of value matching. We now focus on evaluating the performance of the different value matching techniques combined with structural matching. Our experimental setup remains the same (see Section 4.6). Likewise, we perform our measurements in two different environments, the PlanetLab network and a local shared cluster.

For the case of the top-down evaluation method which uses pruning, we run two versions of this method based on how we select the predicate we include for each query in the value filters. The first version uses selectivity statistics to insert the most selective predicate (denoted as *top-down* (most-sel)*) and in the second version we perform a random selection (denoted as *top-down* (random)*).



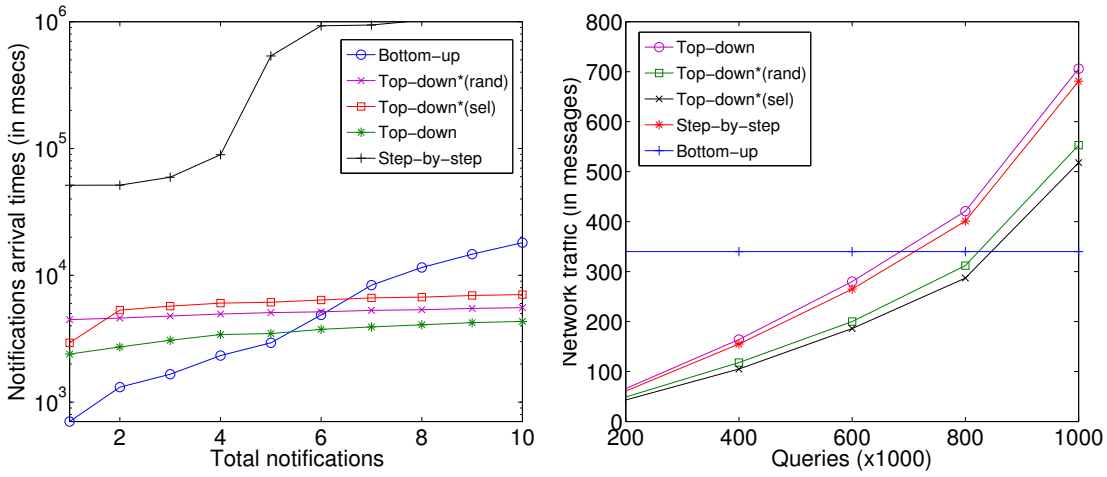
(a) All methods (cluster) (b) All methods (PlanetLab)

Figure 5.11: Filtering latency (different value-matching methods)

5.5.1 Filtering data

In this experiment, we study the performance of our methods during XML filtering. Firstly, we show results from experiments conducted using the cluster machines after indexing 100,000 queries. Figure 5.11(a) shows the filtering latency when queries involve 2 and 4 predicates respectively. We observe that methods which evaluate predicates in a top-down fashion outperform the others by a wide margin. The step-by-step method exhibits the worst performance in both cases, greatly deteriorating as the number of predicates per query is increased from 2 to 4. This is due to the fact that peers spend a lot of effort evaluating predicates for queries whose structure is not matched by the XML data. Likewise, bottom-up evaluation method which indexes queries based on their value-based predicates also demonstrates a poor performance, even though it is 10 times faster compared to step-by-step. This is explained because each peer responsible for a candidate predicate, as this is generated by incoming XML data, performs structural matching for all queries containing this predicate.

We now proceed with the experiments conducted using the Planetlab network. Figure 5.12(a) shows the filtering latency for all different value-matching methods. As we discussed earlier, since Planetlab peers are located in four continents and network delays in such a setting are considerably higher compared to the ones observed in the cluster, filtering requires more time. As before, the step-by-step evaluation method exhibits the worst performance suffering even more from the network delays of Planetlab peers. It is also interesting to study how the system generates the notifications during filtering the XML documents. We depict these measurements in Figure 5.12(a). Bottom-up evaluation method generates notifications faster than the top-down evaluation methods at the beginning of the filtering process. This



(a) Notification arrivals (PlanetLab) (b) Filtering messages

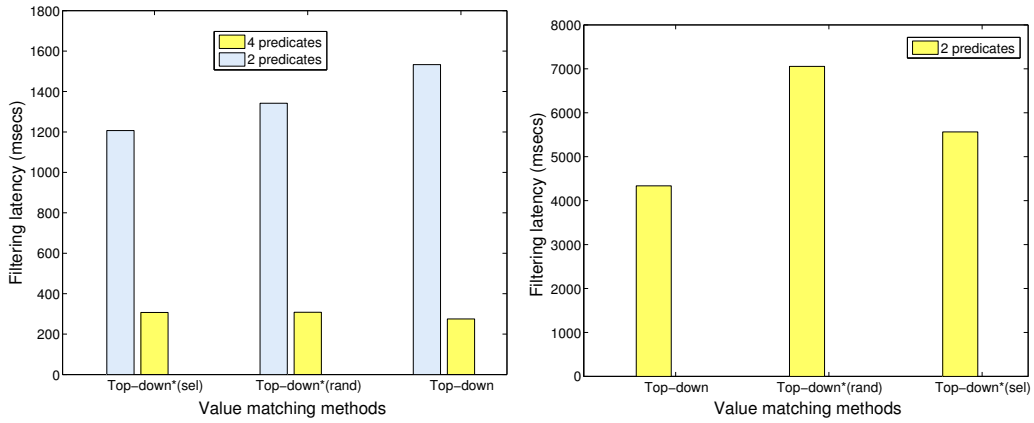
Figure 5.12: Filtering operation

is due to having less peers participating in the filtering process and thus decreasing network delays. However, top-down evaluation methods exhibit a more stable performance since the total amount of processing load is distributed more evenly among a larger number of peers. For example, bottom-up evaluation generates the $260th$ notification about 3 seconds later compared to the top-down evaluation methods.

In Figure 5.12(b), we show the total number of messages that travel through the network during filtering of XML data while varying the total number of indexed queries. Note that the notifications generated during the filtering are not included since all methods generate the same amount of notification messages. Bottom-up method generates a constant number of messages since this is analogous to the number of candidate predicates generated by the XML data and independent of the total indexed queries. Moreover, top-down evaluation method and step-by-step generate almost the same number of messages in all cases. This is due to the fact that both methods traverse almost the same NFA fragment during XML filtering. Finally, when pruning is used less messages are sent since, as expected, execution is stopped at some cases.

5.5.2 Benefit of using value filters

In this experiment, we study the benefit of using Bloom filters to summarize predicate information aiming at stopping NFA execution if we determine that no query can be satisfied by the incoming XML data. We compare our methods after having indexed 100,000 queries and again show notification arrival times after publishing the XML documents. The number of predicates per query ranges from 2 to 4. Figures 5.13(a) and 5.13(b) show the arrival times



(a) Top-down methods (Cluster) (b) Top-down methods (PlanetLab)

Figure 5.13: Benefit of pruning operation

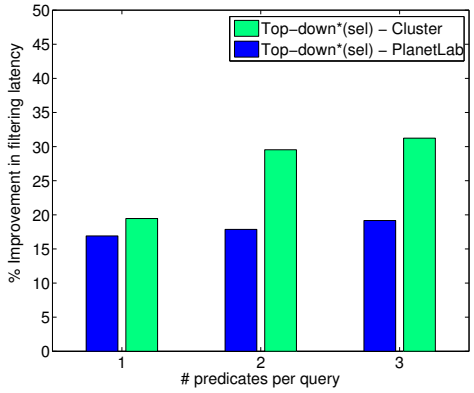


Figure 5.14: Performance improvement

for queries involving 2 and 4 predicates respectively. In general, we observe that top-down evaluation demonstrates a better performance when pruning is used, either by exploiting selectivity statistics or not. As queries involve a larger number of predicates, the gain of using Bloom filters increases, resulting in faster arrival times. This is depicted in Figure 5.13(a) where queries involve 4 predicates and the use of value filters results in a better performance. In addition, the use of selectivity statistics when constructing the value filters results in better arrival times in most cases. We expect that we could further improve its performance with more sophisticated techniques for selectivity estimation but we consider this to be out of the scope of the paper.

Figure 5.14 summarizes how the use of selectivity statistics can result in better performance as we increase the number of predicates per query. We have conducted this experiment in both Planetlab and the cluster. We measure the performance improvement using the for-

mula $\frac{t_{base}-t_{method}}{t_{base}}$, where t_{method} is the filtering time for the method under observation and t_{base} is the filtering time of the top-down evaluation method without any pruning. As shown in the figure, for queries with one predicate, the use of pruning results in a 20% improvement in Planetlab, while in the cluster, where network delays are minimized, we achieve a 30% improvement. As the number of predicates per query grows, the performance improvement due to the use of pruning over the simple top-down evaluation method increases significantly. We have experimented with a higher number of predicates using other datasets and the trend observed was similar. However, in most datasets, more predicates result in significantly less notifications, minimizing the potential improvement in performance and the importance of the measurements.

5.6 Summary

In this chapter, we described methods for the combined structural and value XML filtering in the distributed environment of FoXtrot. One of our methods utilizes Bloom filters to summarize predicate information and decrease the effort spent during value matching. We experimentally evaluated our approach on both PlanetLab and on a local cluster and demonstrated how our methods scale in both the size of query set and the number of predicates per query.

This chapter concludes our presentation of the research conducted in terms of this thesis. In the next chapter, we sum up by highlighting our main contributions. We also discuss open problems and possible directions for future work.

Chapter 6

Conclusions

We conclude this work by presenting a short summary of the research conducted in this thesis, highlighting our main contributions and discussing possible directions for future research.

6.1 Summary

As the Web is growing continuously, a great amount of data becomes available to users, making it more difficult for them to discover interesting information by searching. As a result, publish/subscribe systems have emerged in recent years as a promising paradigm. In this thesis, we described an XML-based publish/subscribe system called FoXtrot built on top of a structured overlay network. FoXtrot is a fully-distributed system for efficient filtering of XML data on very large sets of XPath queries. To achieve this, we utilize the successful automata-based XML filtering engine YFilter, distribute the automaton among the network peers and design methods that exploit the inherent parallelism of an NFA. This way different peers participate in the filtering process by executing in parallel several paths of the NFA.

We show that our approach overcomes the weaknesses of typical content-based XML dissemination systems built on top of meshes or tree-based overlays while paying special attention to load balancing. The design of FoXtrot allows us to employ simple yet effective replication methods for achieving a balanced load distribution. In addition, replication also improves FoXtrot's resilience to faults by introducing redundancy.

The majority of previous approaches addressed the XML filtering problem focusing on designing highly efficient structural matching techniques both in centralized and distributed environments. However, the need for more sophisticated predicate evaluation techniques has also become evident. Especially when queries contain a large number of predicates, the computation time can be dominated by predicate evaluation. In FoXtrot, we combine the

strength of the NFA structure for efficiently matching XPath queries along with different methods for evaluating value-based predicates distributing the load for both tasks among the network peers.

We have implemented our system FoXtrot and performed an extensive experimental evaluation under various conditions. Our evaluation was done in a controlled environment of a local cluster and on the worldwide testbed provided by the PlanetLab network. We demonstrated that FoXtrot can index millions of user queries achieving a high indexing and filtering throughput. At the same time FoXtrot exhibits very good load balancing properties and is also scalable with respect to network size since it improves its performance as we add more peers to the network.

6.2 Future directions

In this section we present a discussion on open problems related with the research conducted in this thesis and give possible future directions for enhancing this work.

6.2.1 Richer data models

As subject of future work, an interesting direction would be to study how our techniques can be applied for richer data models like the case of the Resource Description Framework (RDF) [63]. The RDF data model uses a simple and abstract knowledge representation to describe Web resources. The main idea for enriching our methods to support such a model lies in the following observation: *Since any path query can be transformed into a regular expression and consequently there exists an NFA for representing this query.* In other words, our techniques described using XML and XPath can be used for other data models and query languages including RDF path queries expressed in a navigational query language like nSPARQL [77]. Pérez et al. [77] study how to construct an NFA for indexing RDF path queries and provide us with the basis to apply our results to RDF. A different approach is proposed by Mo and Yuqing [112] who decompose RDF graphs to XML trees and actually demonstrate improved query processing performance compared to existing RDF techniques. Such an approach can also be studied in the context of FoXtrot identifying potential issues that arise. A work that deals with continuous RDF query processing on top of DHTs has been proposed by Liarou et al. [57].

6.2.2 Predicate evaluation

We described our value matching methods focusing on the efficient evaluation of equality predicates. Future work could concentrate on extending these methods for a richer class of predicates like range predicates. In addition, one could consider using more fine-grained statistics for estimating the selectivities of the predicates as well as other a more sophisticated sampling method. For example, Pitoura and Triantafillou [79] focus on distributed sampling algorithms on top of DHTs and describe sampling algorithms for estimating peer loads. These algorithms can be easily adapted to obtain better estimates of predicate selectivities by combining the estimates computed by different peers. For instance, using the random walking algorithm described [79], each peer can collect the estimated selectivities from its neighbors and compute better estimates.

6.2.3 Load balancing

As we discussed, our load balancing method currently depends on the properties of the NFA tree-like structure where the states of lower depths are typically accessed more frequently than the others. Another interesting case would be if the frequencies of visiting the NFA states are not dependent on the depth of the states but follow a different distribution. It would be interesting to demonstrate how FoXtrot can adapt is such situations. One possible direction would be to estimate these frequencies and then design a load balancing method that exploits these statistics for achieving an even load distribution.

6.2.4 Fault tolerance and churn

Since we design FoXtrot as a distributed system expected to run on top of the Internet, we also expect it to continue operating in the presence of failures. We studied how replication-based methods can assist us in increasing the resilience of FoXtrot to failures. A future enhancement of our methods could be to design methods that react on failures trying to minimize the chance of future failures causing information loss in the system.

Apart from dealing with infrequent node failures, almost every distributed system should also address *churn*. Churn refers to the case where the participating network nodes change frequently due to joins, leaves, and failures [37]. This problem has been studied by Rhea et al. [87] for improving the churn resilience of the Bamboo DHT, while Godfrey et al. studied ways for reducing the effects of churn in a distributed system [37]. A possible extension of our work could be to incorporate churn into the system design and study the effect of such membership events on the performance of FoXtrot.

Abbreviations

DHT	Distributed Hash Table
NFA	Non-deterministic Finite Automaton
P2P	Peer-to-Peer
XML	Extensible Markup Language
XPath	XML Path Language

Bibliography

[1] DBLP XML records. http://dblp.uni-trier.de/xml/.

[2] Gnutella development forum. `http://groups.yahoo.com/group/the_gdf/`.

[3] An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 262–, Washington, DC, USA, 1999. IEEE Computer Society.

[4] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a self-organizing structured P2P system. *SIGMOD Rec.*, 32(3):29–33, 2003.

[5] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*, pages 606–615, Washington, DC, USA, 2008. IEEE Computer Society.

[6] I. Aekaterinidis and P. Triantafillou. PastryStrings: A comprehensive content-based publish/subscribe DHT network. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, pages 23–, Washington, DC, USA, 2006. IEEE Computer Society.

[7] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[8] James Aspnes, Zoë Diamadi, and Gauri Shah. Fault-tolerant routing in peer-to-peer systems. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 223–232, New York, NY, USA, 2002. ACM.

[9] James Aspnes and Gauri Shah. Skip graphs. *ACM Trans. Algorithms*, 3, November 2007.

[10] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Commun. ACM*, 46:43–48, February 2003.

[11] D. Barbosa, L. Mignet, and P. Veltri. Studying the XML Web: Gathering statistics

from an XML sample. *World Wide Web*, 9(2):187–212, 2006.

[12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[13] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. World Wide Web Consortium, Recommendation, December 2010. `http://www.w3.org/TR/2010/REC-xquery-20101214/`.

[14] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath Lookup Queries in P2P Networks. In *Proceedings of the 6th annual ACM international workshop on Web information and data management (WIDM '04)*, pages 48–55, New York, NY, USA, 2004. ACM.

[15] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0. World Wide Web Consortium, Recommendation, November 2008. `http://www.w3.org/TR/2008/REC-xml-20081126/`.

[16] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *Proceedings. 19th International Conference on Data Engineering (ICDE '03)*, pages 139–150, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[17] K. Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Junichi Tatemura, and Divyakant Agrawal. Afilter: adaptable xml filtering with prefix-caching suffix-clustering. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 559–570. VLDB Endowment, 2006.

[18] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19:332–383, August 2001.

[19] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized publish-subscribe infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8), October 2002. Special issue on Network Support for Multicast Communications.

[20] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*, page 235, Washington, DC, USA, 2002. IEEE Computer Society.

[21] C. Y. Chan and Y. Ni. Efficient XML Data Dissemination with Piggybacking. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)*, pages 737–748, New York, NY, USA, 2007. ACM Press.

[22] R. Chand and P. Felber. Scalable Distribution of XML Content with XNet. *IEEE*

Transactions on Parallel and Distributed Systems, 19(4):447–461, 2008.

[23] R. Chand, P. Felber, and M. Garofalakis. Tree-pattern similarity estimation for scalable content-based routing. *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1016–1025, April 2007.

[24] R. Chand and P. A. Felber. A Scalable Protocol for Content-Based Routing in Overlay Networks. In *Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA '03)*, pages 123–, Washington, DC, USA, 2003. IEEE Computer Society.

[25] James Clark and Steven J. DeRose. XML Path Language (XPath) Version 1.0. World Wide Web Consortium, Recommendation, November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116/`.

[26] M. P. Consens and T. Milo. Optimizing queries on files. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data (SIGMOD '94)*, pages 301–312, New York, NY, USA, 1994. ACM.

[27] Mariano P. Consens and Tova Milo. Algebras for querying text regions (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 11–22, New York, NY, USA, 1995. ACM.

[28] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34:56–78, February 1991.

[29] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *Proceedings of the Thirtieth international conference on Very large data bases (VLDB '04)*, pages 612–623. VLDB Endowment, 2004.

[30] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM TODS*, 28(4):467–516, 2003.

[31] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.

[32] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[33] P. Felber, C.Y. Chan, M. Garofalakis, and R. Rastogi. Scalable filtering of XML data for Web services. *IEEE Internet Computing*, 7(1):49–57, 2003.

[34] W. Fenner, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang. XTreeNet: Scalable overlay networks for XML content dissemination and querying (synopsis). In *Proceedings of the 10th International Workshop on Web Content Caching and Distribution (WCW '05)*, pages 41–46, Washington, DC, USA, 2005. IEEE Com-

puter Society.

[35] FreePastry 2.1 release, 2009. `http://www.freepastry.org/FreePastry/`.

[36] L. Galanis, Y. Wang, S. Jeffery, and D. J. DeWitt. Locating data sources in large distributed systems. In *Proceedings of the 29th international conference on Very large data bases (VLDB '03)*, pages 874–885. VLDB Endowment, 2003.

[37] P. Brighten Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *Proceedings of the Annual ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2006.

[38] X. Gong, W. Qian, Y. Yan, and A. Zhou. Bloom filter-based XML packets filtering for millions of path queries. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 890–901, Washington, DC, USA, 2005. IEEE Computer Society.

[39] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[40] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, pages 173–189, London, UK, 2002. Springer-Verlag.

[41] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03)*, pages 419–430, New York, NY, USA, 2003. ACM.

[42] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[43] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[44] S. Hou and H. A. Jacobsen. Predicate-based filtering of XPath expressions. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, pages 53–, Washington, DC, USA, 2006. IEEE Computer Society.

[45] IBM. Gryphon: Publish/subscribe over public networks. Technical report, 2001.

[46] IBM XML Generator, 1999. `http://www.alphaworks.ibm.com/xmlgenerator`.

[47] Zachary G. Ives, A. Y. Halevy, and D. S. Weld. An xml query engine for network-bound data. *The VLDB Journal*, 11:380–402, December 2002.

[48] H. V. Jagadish, B. C. Ooi, K. Tan, and Q. H. Vu. Baton: a balanced tree structure

for peer-to-peer networks. In *Proceedings of the 31st international conference on Very large data bases (VLDB '05)*, pages 661–672. VLDB Endowment, 2005.

[49] H. V. Jagadish, B. C. Ooi, K. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD '06)*, pages 1–12, New York, NY, USA, 2006. ACM.

[50] Jayanthkumar Kannan, Beverly Yang, Scott Shenker, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Sung ju Lee. Smartseer: Using a dht to process continuous queries over peer-to-peer networks. In *IEEE INFOCOM*, 2006.

[51] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[52] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04*.

[53] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *Advances in Database Technology - EDBT 2004*, pages 29–47. Springer Berlin / Heidelberg, 2004.

[54] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. In *Proceedings of the 31st international conference on Very large data bases (VLDB 2005)*, 2005.

[55] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. Value-based Predicate Filtering of XML Documents. *Data and Knowledge Engineering*, 67(1):51–73, 2008.

[56] Laks V. S. Lakshmanan and Sailaja Parthasarathy. On efficient matching of streaming xml documents and queries. In *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '02, pages 142–160, London, UK, 2002. Springer-Verlag.

[57] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Evaluating conjunctive triple pattern queries over large structured overlay networks. In *International Semantic Web Conference*, pages 399–413, 2006.

[58] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. LH*a scalable, distributed data structure. *ACM Trans. Database Syst.*, 21:480–525, December 1996.

[59] Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 395–406, New York, NY, USA, 2003.

ACM.

[60] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys Tutorials, IEEE*, 7(2):72 – 93, quarter 2005.

[61] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A transducer-based xml query processor. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 227–238. VLDB Endowment, 2002.

[62] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 183–192, New York, NY, USA, 2002. ACM.

[63] F. Manola and E. Miller. RDF primer: W3c recommendation. *Decision Support Systems*, 2004.

[64] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, 2002. Springer-Verlag.

[65] David Megginson. SAX: the Simple API for XML. `http://www.saxproject.org/`.

[66] I. Miliaraki, Z. Kaoudi, and M. Koubarakis. XML data dissemination using automata on top of structured overlay networks. In *Proceedings of the 17th International World Wide Web Conference (WWW '08)*, pages 865–874, New York, NY, USA, 2008. ACM.

[67] I. Miliaraki and M. Koubarakis. Distributed structural and value XML filtering. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS '10)*, pages 2–13, New York, NY, USA, 2010. ACM.

[68] Iris Miliaraki and Manolis Koubarakis. Foxtrot: Distributed structural and value xml filtering.

[69] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early profile pruning on XML-aware publish/subscribe systems. In *Proceedings of the 33rd international conference on Very large data bases (VLDB '07)*, pages 866–877. VLDB Endowment, 2007.

[70] Mirella M. Moro, Petko Bakalov, and Vassilis J. Tsotras. Early profile pruning on xml-aware publish-subscribe systems. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 866–877. VLDB Endowment, 2007.

[71] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. Federal Information Processing Standards Publication, Gaithersburg, MD, USA, apr 1995.

[72] O. Papaemmanouil and U. Cetintemel. SemCast: Semantic multicast for content-based data dissemination. In *Proceedings of the 21st International Conference on*

Data Engineering (ICDE '05), pages 242–253, Washington, DC, USA, 2005. IEEE Computer Society.

[73] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03)*, pages 431–442, New York, NY, USA, 2003. ACM.

[74] Feng Peng and Sudarshan S. Chawathe. Xpath queries on streaming data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 431–442, New York, NY, USA, 2003. ACM.

[75] Feng Peng and Sudarshan S. Chawathe. Xsq: A streaming xpath engine. *ACM Trans. Database Syst.*, 30:577–623, June 2005.

[76] João Pereira, Françoise Fabret, François Llirbat, Radu Preotiuc-Pietro, Kenneth A. Ross, and Dennis Shasha. Publish/subscribe on the web at extreme speed. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 627–630, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[77] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *Web Semant.*, 8:255–270, November 2010.

[78] Milenko Petrovic, Haifeng Liu, and Hans-Arno Jacobsen. G-topss: fast filtering of graph-based metadata. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 539–547, New York, NY, USA, 2005. ACM.

[79] T. Pitoura and P. Triantafillou. Load Distribution Fairness in P2P Data Management Systems. In *ICDE 2007*.

[80] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Saturn: Range queries, load balancing and fault tolerance in dht data systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2010.

[81] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 311–320, New York, NY, USA, 1997. ACM.

[82] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: prefix hash tree. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC '04)*, pages 368–368, New York, NY, USA, 2004. ACM.

[83] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gün Sirer. Corona: a high performance publish-subscribe system for the World Wide Web. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[84] P. R. Rao and B. Moon. Locating XML documents in a peer-to-peer network using

distributed hash tables. *IEEE Trans. on Knowl. and Data Eng.*, 21(12):1737–1752, 2009.

[85] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[86] Sean Rhea, Byung-Gon Chun, John Kubiatowicz, and Scott Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *Proceedings of the 2nd conference on Real, Large Distributed Systems - Volume 2*, WORLDS'05, pages 25–30, Berkeley, CA, USA, 2005. USENIX Association.

[87] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[88] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, pages 329–350, London, UK, 2001. Springer-Verlag.

[89] Thomas Schwentick. Automata for xml–a survey. *Journal of Computer and System Sciences*, 73(3):289 – 315, 2007. Special Issue: Database Theory 2004.

[90] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *PODS*, pages 53–64. ACM Press, 2002.

[91] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[92] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using XML. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP '01)*, pages 160–173, New York, NY, USA, 2001. ACM.

[93] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, pages 149–160, New York, NY, USA, 2001. ACM.

[94] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11:17–32, February 2003.

[95] Sonesh Surana, Brighten Godfrey, Karthik Lakshminarayanan, Richard Karp, and Ion Stoica. Load balancing in dynamic structured peer-to-peer systems. *Perform. Eval.*, 63(3):217–240, 2006.

[96] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, DEBS '03, pages 1–8, New York, NY, USA, 2003. ACM.

[97] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. World Wide Web Consortium, Recommendation, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`.

[98] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing a scalable XML publish/subscribe system using relational database systems. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD '04)*, pages 479–490, New York, NY, USA, 2004. ACM.

[99] Peter Triantafillou and Ioannis Aekaterinidis. Content-based publish-subscribe over structured p2p networks. In *Proc. third Int. Workshop Distributed Event-based Systems (DEBS"04)*, pages 24–25, 2004.

[100] Christos Tryfonopoulos, Stratos Idreos, and Manolis Koubarakis. Publish/subscribe functionality in IR environments using structured overlay networks. In *SIGIR*, pages 322–329, 2005.

[101] Dimitrios Tsoumakos and Nick Roussopoulos. A comparison of peer-to-peer search methods. In *WebDB*, pages 61–66, 2003.

[102] H. Uchiyama, M. Onizuka, and T. Honishi. Distributed xml stream filtering system with high scalability. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 968–977, Washington, DC, USA, 2005. IEEE Computer Society.

[103] Z. Vagena, M. M. Moro, and V. J. Tsotras. Value-aware RoXSum: Effective message aggregation for XML-aware information dissemination. In *10th International Workshop on the Web and Databases (WebDB '07)*, 2007.

[104] Jeffrey Scott Vitter. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[105] Bruce Watson. Practical optimizations for automata. In Derick Wood and Sheng Yu, editors, *Automata Implementation*, volume 1436 of *Lecture Notes in Computer Science*, pages 232–240. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0031396.

[106] XMark: An XML benchmark project, 2001. `http://www.xml-benchmark.org/`.

[107] YFilter 1.0 release, 2004. `http://yfilter.cs.umass.edu/code_release.htm`.

[108] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *Peer-to-Peer Systems IV, 4th International Workshop, IPTPS 2005*, volume 3640 of *Lecture Notes in Computer Science*, pages 47–57. Springer, 2005.

[109] Xi Zhang, Liang Huai Yang, Mong-Li Lee, and Wynne Hsu. Scaling sdi systems via

query clustering and aggregation. In *DASFAA*, pages 208–219, 2004.

[110] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA, 2001.

[111] A. Zhou, W. Qian, X. Gong, and M. Zhou. Sonnet: An efficient distributed content-based dissemination broker (poster paper). In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)*, pages 1094–1096, New York, NY, USA, 2007. ACM.

[112] Mo Zhou and Yuqing Wu. XML-based RDF data management for efficient query processing. In *Proceedings of the 13th International Workshop on the Web and Databases*, WebDB '10, pages 3:1–3:6, New York, NY, USA, 2010. ACM.

[113] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '01, pages 11–20, New York, NY, USA, 2001. ACM.