Preference-Aware Publish/Subscribe Delivery with Diversity

Marina Drosou

MASTER THESIS

— ♦ —

Ioannina, September 2008

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF IOANNINA

Σύστημα Έκδοσης/Συνδρομής για τη Μετάδοση Πληροφοριών με βάση τις Προτιμήσεις και την Ποικιλομορφία

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από την

Μαρίνα Δρόσου

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Σεπτέμβριος 2008

Dedication

To my parents, and Gregory...

Acknowledgements

I would first like to thank my supervisor, Prof. Evaggelia Pitoura, for her help, dedication and time spent during the elaboration of this thesis. I would also like to thank all the people of the DMOD laboratory who turned my time there into a joyful experience, especially my office-mate and friend Kostas Stefanidis and also Eftychia Baikoussi, Mirto Ntetsika, Kostas Lillis and Prof. Panos Vassiliadis. Finally, I would like to thank my parents and my friends for their continuous support throughout all the years of my studies.

Table of Contents

List of Figures

List of Tables

Algorithm Index

Abstract

Marina K. Drosou. MSc, Computer Science Department, University of Ioannina, Greece. September, 2008. *Preference-Aware Publish/Subscribe Delivery with Diversity.* Thesis Supervisor: Evaggelia Pitoura.

In publish/subscribe systems, users describe their interests in specific events via subscriptions and get notified whenever new events that match their interests become available. Typically, in such systems, all subscriptions are considered equally important. As the amount of information generated increases rapidly, to control the amount of data delivered to users, we propose enhancing publish/subscribe with a ranking mechanism based on user preferences, so that only top-ranked events are delivered to each user. Ranking is based on letting users express their preferences on events by ordering the associated subscriptions. Since many times top-ranked events are similar to each other, we propose diversifying delivered events to further increase user satisfaction. Furthermore, we examine a number of different timing policies for delivering ranked events to users. We have fully implemented our approach in SIENA, a popular publish/subscribe middleware system, and report experimental results of its deployment.

Εκτεταμενη Περιληψη

Μαρίνα Δρόσου του Κωνσταντίνου και της Κωνσταντίνας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Σεπτέμβριος, 2008. *Σύστημα Έκδοσης/Συνδρομής για τη Μετάδοση Πληροφοριών με βάση τις Προτιμήσεις και την Ποικιλομορφία*. Επιβλέπουσα: Ευαγγελία Πιτουρά.

Στα συστήματα έκδοσης/συνδρομής, οι συνδρομητές εκφράζουν τα ενδιαφέροντά τους για διάφορες κατηγορίες γεγονότων και στη συνέχεια ενημερώνονται από το σύστημα μόλις δημοσιεύονται γεγονότα που ταιριάζουν στα ενδιαφέροντά τους. Καθώς ο όγκος της πληροφορίας που παράγεται αυξάνεται ραγδαία, προτείνουμε την επέκταση των συστημάτων έκδοσης/συνδρομής με ένα μηχανισμό διάταξης, έτσι ώστε μόνο τα πιο ενδιαφέροντα γεγονότα να παραδίδονται στους χρήστες. Για να διατάξουμε τα γεγονότα, επιτρέπουμε στους χρήστες να ορίζουν προτιμήσεις ανάμεσα στα ενδιαφέροντά τους. Συνδέουμε κάθε γεγονός με ένα χρόνο λήξης έτσι ώστε πολύ ενδιαφέροντα γεγονότα να μην αποκλείουν νεότερα από το να παραδίδονται στους χρήστες. Για να αυξήσουμε περαιτέρω την ικανοποίηση των χρηστών, προτείνουμε την αύξηση της ποικιλομορφίας των γεγονότων που παραδίδονται. Για να γίνει αυτό, επιλέγουμε τα γεγονότα που θα παραδοθούν λαμβάνοντας υπόψη όχι μόνο την σχέση τους με τα ενδιαφέροντα των χρηστών αλλά και την μεταξύ τους ομοιότητα.

Πιο αναλυτικά, στα παραδοσιακά συστήματα έκδοσης/συνδρομής, όλες οι συνδρομές των χρηστών θεωρούνται εξίσου σημαντικές. Αυτή η υπόθεση περιορίζει τις επιλογές του χρήστη. Για παράδειγμα, ας υποθέσουμε έναν χρήστη ο οποίος ενδιαφέρεται γενικά για δραματικές ταινίες. Πιο συγκεκριμένα, ας υποθέσουμε ότι ενδιαφέρεται περισσότερο για δραματικές ταινίες του T. Burton από ότι για δραματικές ταινίες του S. Spielberg. Στην περίπτωση αυτή, ο χρήστης θα ήθελε να ειδοποιείται για δραματικές ταινίες του S. Spielberg μόνο στην περίπτωση που δεν υπάρχουν αρκετές πληροφορίες για δραματικές ταινίες του T. Burton. Τα υπάρχοντα συστήματα έκδοσης/συνδρομής όμως, δεν παρέχουν τη δυνατότητα έκφρασης αυτής της προτίμησης.

Στην εργασία αυτή, προτείνουμε τη χρήση κάποιου είδους διάταξης μεταξύ των ενδιαφερόντων των χρηστών, έτσι ώστε να τους παρέχεται η δυνατότητα να εκφράσουν το γεγονός πως κάποια ενδιαφέρονται είναι πιο σημαντικά από κάποια άλλα. Για να διατάξουμε τα ενδιαφέροντα, χρησιμοποιούμε προτιμήσεις. Διάφορα μοντέλα προτιμήσεων έχουν προταθεί στο παρελθόν. Τα περισσότερα ακολουθούν είτε την ποσοτική, είτε την ποιοτική προσέγγιση. Στην πρώτη περίπτωση οι χρήστες χρησιμοποιούν μαθηματικές συναρτήσεις που αναθέτουν έναν αριθμητικό βαθμό σε κάθε δεδομένο, ενώ στη δεύτερη οι προτιμήσεις

μεταξύ των διάφορων δεδομένων ορίζονται άμεσα με τη χρήση δυαδικών σχέσεων. Για να εκφράσουμε προτιμήσεις μεταξύ συνδρομών, πρώτα εισάγουμε τις συνδρομές προτίμησης. Οι συνδρομές προτίμησης μπορούν να χρησιμοποιηθούν και με τα δύο μοντέλα προτιμήσεων με το να συνδέουν έναν βαθμό ενδιαφέροντος σε κάθε συνδρομή.

Βασισμένοι στις συνδρομές προτίμησης, εισάγουμε μία παραλλαγή του προτύπου έκδοσης/συνδρομής στην οποία οι χρήστες λαμβάνουν μόνο τα k πιο ενδιαφέροντα γεγονότα και όχι όλα όσα ταιριάζουν στις συνδρομές τους. Εξετάσουμε έναν αριθμό πολιτικών χρονισμού για την παράδοση γεγονότων: (i) την πολιτική συνεχούς ροής, (ii) την περιοδική πολιτική και (iii) την πολιτική κυλιόμενου παραθύρου. Στην πολιτική συνεχούς ροής, τα πιο ενδιαφέροντα γεγονότα προωθούνται στους χρήστες τη στιγμή της έκδοσής τους. Στην περιοδική πολιτική, τα πιο ενδιαφέροντα γεγονότα προωθούνται ανά τακτά χρονικά διαστήματα, ενώ στην πολιτική κυλιόμενου παραθύρου συνδυάζουμε τις δύο προηγούμενες προσεγγίσεις με το να υπολογίζουμε και να προωθούμε τα πιο ενδιαφέροντα γεγονότα ενός κυλιόμενου παραθύρου γεγονότων με προκαθορισμένο μήκος.

Τα πιο ενδιαφέρονται γεγονότα για κάθε χρήστη συχνά παρουσιάζουν μεγάλη ομοιότητα. Οστόσο, η αύξηση της ποικιλομορφίας των γεγονότων που παραδίδονται τελικά στους χρήστες αυξάνει την ικανοποίησή τους από το σύστημα. Για παράδειγμα, ο χρήστης του παραδείγματός μας θα ήθελε να λαμβάνει πληροφορίες κυρίως για διαφορετικές δραματικές ταινίες του T. Burton και, μια στο τόσο, και για κάποιες δραματικές ταινίες του S. Spielberg. Για το λόγο αυτό, τροποποιούμε το μηχανισμό υπολογισμού των k καλύτερων γεγονότων έτσι ώστε να λαμβάνουμε υπόψη και την ποικιλομορφία τους. Σε αυτήν την κατεύθυνση, χρησιμοποιούμε ένα συνδυασμό της σημασίας του κάθε γεγονότος για τον εκάστοτε χρήστη και της ομοιότητάς του με άλλα σημαντικά γεγονότα για αυτόν. Εξετάζουμε πως μπορούμε να αυξήσουμε την ποικιλομορφία των γεγονότων για κάθε πολιτική χρονισμού.

Για τον αποδοτικό εντοπισμό των συνδρομών προτίμησης που ταιριάζουν με ένα νέο γεγονός, υιοθετούμε μία αναπαράσταση των συνδρομών βασισμένη σε γράφους, την οποία καλούμε γράφο συνδρομών προτίμησης, και την χρησιμοποιούμε για την προώθηση των γεγονότων που δημοσιεύονται στους χρήστες.

Υλοποιήσαμε ένα πρότυπο σύστημα, το οποίο ονομάζουμε PrefSIENA, το οποίο επεκτείνει το σύστημα SIENA, ένα δημοφιλές σύστημα έκδοσης/συνδρομής, με την ενσωμάτωση συνδρομών προτίμησης, πολιτικών χρονισμού και αύξηση ποικιλομορφίας των γεγονότων με σκοπό τον εντοπισμό των k πιο σημαντικών γεγονότων για κάθε χρήστη και την παράδοσή τους σε αυτόν. Παρουσιάζουμε έναν αριθμό πειραματικών αποτελεσμάτων σχετικά με τον αριθμό και την ποιότητα των γεγονότων που παραδίδονται στους χρήστες από το PrefSIENA σε σύγκριση με το σύστημα SIENA, καθώς και το επιπλέον κόστος που απαιτείται για την διάταξη των γεγονότων.

Chapter 1

Introduction

1.1 Scope of Thesis

With the explosion of the amount of information that is currently available online, publish/subscribe systems offer an attractive alternative to searching by providing a proactive model of information supply. In such systems, users express their interest in specific pieces of data, or *events*, via *subscriptions*. Then, they are *notified* whenever some other user generates (or *publishes*) an event that *matches* one of their subscriptions. Typically, all subscriptions are considered equally important and users are notified whenever a published event matches any of their subscriptions.

However, getting notified about all matching events may lead to overwhelming the users with huge amounts of notifications about events, thus hurting the acceptability of publish/subscribe systems. To control the rate of notifications received by the subscribers, it would be useful to allow them to rank the importance or relevance of events. Then, they would only receive notifications for the most important or relevant among them. For example, take a user, say John, that generally likes drama movies but prefers drama movies directed by T. Burton to drama movies directed by S. Spielberg. Ideally, John would like to receive notifications about S. Spielberg drama movies only if there are no, or not enough, notifications about T. Burton drama movies.

In this work, we propose enhancing publish/subscribe with a ranking mechanism based on user preferences, so that only top-ranked events are delivered to each user. To do this, we extend subscriptions to allow users express the fact that some events are more important or relevant to them than others. To indicate priorities among subscriptions, we

introduce *preferential subscriptions*. In general, there are two basic approaches to specifying preferences among data items: the quantitative and the qualitative approach. In the *quantitative* approach (e.g. [6, 20, 27]), users employ scoring functions that associate a numeric score with specific data items to indicate their interest in them. In the *qualitative* approach (e.g. [10, 15, 19]), preferences between two data items are specified directly, typically using binary preference relations. We show how to formulate preferences among subscriptions using each one of these approaches. Published events are ranked so that an event that matches a highly preferred subscription is ranked higher than an event that matches a subscription with a lower preference.

Based on preferential subscriptions, we introduce a top-k variation of publish/subscribe in which users receive only the matching events having the k highest ranks as opposed to all events matching their subscriptions. Since the delivery of events is continuous, we also introduce a number of timing policies that determine the range of events over which the top-k computation is performed.

However, the top-k events are often very similar to each other. Besides pure accuracy achieved by matching the criteria set by the users, diversification, i.e. recommending items that differ from each other, has been shown to increase user satisfaction [29]. For instance, our user John would probably like to receive information about different drama movies by T. Burton as well as a couple of S. Spielberg movies once in a while. To this end, we adjust the top-k computation to take also into account the *diversity* of the delivered events. To achieve this, we consider both the importance of each event, as specified by the user preferences, as well as its diversity from the other top-ranked events. We examine how the results can be diversified for each of the timing policies.

As a proof-of-concept, we have implemented a prototype, termed PrefSIENA [3]. PrefSIENA extends SIENA [4], a popular publish/subscribe middleware system, with preferential subscriptions, timing policies and diversity towards achieving top-k event delivery. We present a number of experimental results that evaluate the number of events delivered by PrefSIENA with respect to the original SIENA system, as well as the rank, freshness and diversity of such events. We also report on the overheads of supporting top-k delivery.

1.2 Thesis Outline

The rest of this thesis is structured as follows. In Chapter 2, we present publish/subscribe preliminaries and introduce preferential subscriptions, i.e. subscriptions augmented with degrees of interest. We also show how to compute the importance of published events for each user. In Chapter 3, we examine a number of different timing policies for delivering events and in Chapter 4, we focus on how to diversify the top-ranked events. In Chapter 5, we describe preferential subscription graphs and introduce an algorithm for computing top-ranked events based on preferential subscriptions. In Chapter 6, we present our evaluation setup and experimental results. Chapter 7 describes related work and finally, Chapter 8 concludes this thesis with a summary of our contributions and outlines future work.

Chapter 2

Preference Model

In this chapter, we first present some background on publish/subscribe systems and describe a typical form of events and subscriptions used in such systems. Then, we introduce an extended version of subscriptions that include the notion of preference. Finally, we examine how to compute the importance of published events for each user.

2.1 Publish/Subscribe Preliminaries

A publish/subscribe system is an event-notification service designed to be used over large-scale networks, such as the Internet. Such systems offer an attractive alternative to searching by providing a proactive model of information supply. Generators of events, called publishers, can publish events to the service and consumers of such events, called subscribers, can subscribe to the service to receive a portion of them. Publishers can publish events at any time. The events will be delivered to all interested subscribers at some point in the future.

Architecture: In general, a publish/subscribe system [11] consists of three main parts: (i) the *publishers* that provide events to the system, (ii) the *subscribers* that consume these events and (iii) an *event-notification service* that stores the various subscriptions, matches the incoming events with them and delivers these events to the appropriate subscribers. As shown in Figure 2.1, the event-notification service provides a number of primitive operations to the users. The **publish**() operation is called by a publisher whenever it wishes to generate a new event. The **subscribe**() operation is called by a subscriber whenever

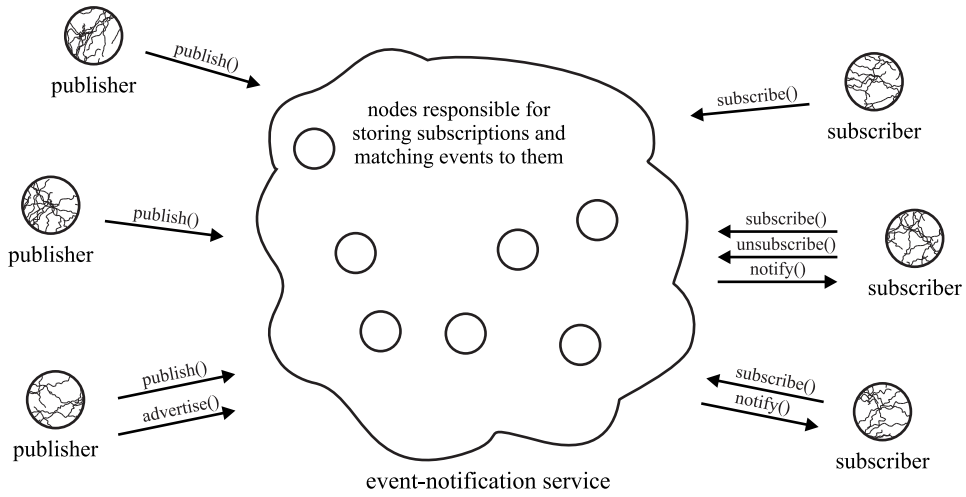


Figure 2.1: Publish/subscribe system operations.

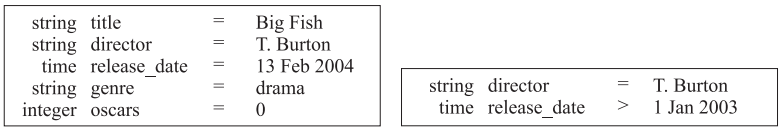
the subscriber wishes to express a new interest. An **unsubscribe**() operation is usually also provided to cancel previous subscriptions. An optional (**un**)**advertise**() operation may be available to publishers, so that they can advertise the content of their future publications. The event-notification service can use the **notify**() operation whenever it wants to deliver an event to a subscriber. An event-notification service can be implemented using either a centralized or a distributed architecture, that is, we may have one or a set of servers responsible for the process of matching events with subscriptions.

Events: We use a generic way to form events, similar to the one used in [7, 13], to maximize user expressiveness. In particular, events are sets of typed attributes. Each event consists of an arbitrary number of attributes and each attribute has a type, a name and a value. Attribute types belong to a predefined set of primitive types, such as "integer" or "string". Attribute names are character strings that take values according to their type. An example event about a movie is shown in Figure 2.2a. Formally:

Definition 2.1 (Event)**.** An event e is a set of typed attributes $\{a_1, \ldots, a_p\}$, where each a_i is of the form $(a_i.type\ a_i.name = a_i.value)$, $1 \leq i \leq p$.

Subscriptions: Subscriptions are used to specify the kind of events users are interested in. Therefore, they can be thought of as filters that are used to filter out all irrelevant information. Each subscription consists of a set of constraints on the values of specific attributes. Each attribute constraint has a type, a name, a binary operator and a value. Types, names and values have the same form as in events. Binary operators include common operators such as $=, \neq, <, >, \leq, \geq$, $substring$, $prefix$ and $suffix$. An example subscription is depicted in Figure 2.2b. Formally:

Definition 2.2 (Subscription)**.** A subscription s is a set of attribute constraints $\{b_1, \ldots, b_q\}$, where each b_i is of the form $(b_i.type\ b_i.name\ \theta_{b_i}\ b_i.value)$, $\theta_{b_i} \in \{=, \neq <, >, \leq, \geq, substring, prefix, suffix\}$, $1 \leq i \leq q$.



(a) Event example. (b) Subscription example.

Figure 2.2: Event and subscription examples.

Matching events with subscriptions: To deliver the appropriate events to subscribers, the event-notification service has to identify all interesting information for each user. To do this, the *Cover Relation* is used. Intuitively, we can say that a subscription s *covers* an event e, or alternatively e *matches* s, if and only if, every attribute constraint of s is satisfied by some attribute of e. Formally:

Definition 2.3 (Cover Relation)**.** Given an event e of the form $\{a_1, \ldots, a_p\}$ and a subscription s of the form $\{b_1, \ldots, b_q\}$, s covers e, if and only if, $\forall\, b_j \in s$, $\exists\, a_i \in e$, such that, $a_i.type = b_j.type$, $a_i.name = b_j.name$ and $((a_i.value)\ \theta_{b_j}\ (b_j.value))$ holds, $1 \leq i \leq p$, $1 \leq j \leq q$.

An event e is delivered to a user, if and only if, the user has submitted at least one subscription s, such that, s covers e. For example, the subscription of Figure 2.2b covers the event of Figure 2.2a and therefore, this event should be delivered to all users who have submitted this subscription.

2.2 Preferential Subscriptions

To allow users to express the fact that some events are *more important* to them than others, we propose extending the publish/subscribe paradigm to incorporate ranking. Our goal is for each subscriber not to receive all matching events but instead, the most interesting of them, i.e. the most highly ranked ones. To achieve this, we allow users to express preferences through *preferential subscriptions*. Preferential subscriptions are subscriptions enhanced with degrees of interest, called *preference ranks*.

Definition 2.4 (Preferential Subscription)**.** A preferential subscription ps_i^X of user X is a pair of the form $ps_i^X = (s_i, prefrank_i^X)$, where s_i is a subscription and $prefrank_i^X$ is a real number in $[0, 1]$ that expresses the degree of interest of X for s_i.

The higher the preference rank, the more interested the user is in events covered by the specified subscription. Examples of preferential subscriptions are shown in Figure 2.3.

In general, preferences can be expressed following either a quantitative or a qualitative approach. Following a *quantitative* preference model, users explicitly provide numeric scores, e.g. values within the range $[0, 1]$, to indicate the importance of each of their subscriptions. In this case, the preference rank that is associated with a subscription is the score provided for it by the user.

string director = T. Burton string genre = drama	0.8

string director = S. Spielberg string genre = drama	0.6

Figure 2.3: Preferential subscription examples.

Following a *qualitative* model, users employ binary relations to define priorities among their subscriptions. Specifically, assume that a user X provides a set of preferential subscriptions P^X and that those preferential subscriptions contain the set of subscriptions S_P^X. To express preferences between subscriptions, X defines *priority conditions* of the form $(s_i \succ s_j)$, s_i, $s_j \in S_P^X$, to denote that s_i is preferred to s_j (e.g. Figure 2.4). Let C^X be the set of priority conditions expressed by user X, i.e. $C^X = \{(s_i \succ s_j) \mid s_i, s_j \in S_P^X\}$. To extract the most preferable subscriptions based on C^X, we use the winnow operator [10]. The first application of the winnow operator returns the set $win^X(1)$ of subscriptions $s_i \in S_P^X$, such that, $\forall s_i \in win^X(1)$ there is no $s_j \in S_P^X$ with $s_j \succ s_i$. To retrieve the most preferable subscriptions after the ones included in $win^X(1)$, we apply the winnow operator a second time. $win^X(2)$ consists of the subscriptions $s_i \in (S_P^X - win^X(1))$, such that, $\forall s_i \in win^X(2)$ there is no subscription $s_j \in (S_P^X - win^X(1))$ with $s_j \succ s_i$. The winnow operator is applied until all user subscriptions are returned.

Generalizing the winnow operator, we introduce a multiple level variation, defined as follows:

Definition 2.5 (Multiple Level Winnow Operator)**.** Assume a user X and the set P^X of the preferential subscriptions of X. Let S_P^X be the set of all subscriptions in P^X and C^X the set of all priority conditions defined by X. The multiple level winnow operator at level l, $l > 1$, returns a set of subscriptions, $win^X(l)$, consisting of the subscriptions $s_i \in S_P^X - \cup_{q=1}^{l-1} win^X(q)$, such that, $\forall\ s_i \in win^X(l)$, $\nexists\ s_j \in S_P^X - \cup_{q=1}^{l-1} win^X(q)$ with $(s_j \succ s_i) \in C^X$.

The preference rank associated with each subscription depends on the winnow level that the subscription belongs to, since subscriptions retrieved earlier are of higher importance for the users. In particular, a subscription $s_i \in win^X(l)$ is associated with a preference rank equal to $1/l$. As in the quantitative model, a higher preference rank indicates a more important subscription.

Any conflicting priorities should be resolved prior to computing preference ranks, either by involving the users or by using some default conflict resolution procedure. Here, we assume that the priorities between subscriptions follow a strict partial order. Therefore, to find the most interesting subscriptions based on C^X, we organize them using a directed graph in which nodes correspond to subscriptions and edges to priority conditions. Then, we apply a topological sort algorithm on this graph to extract the most preferable subscriptions in levels. For example, Figure 2.5 depicts the graph for a set of priority conditions and the extracted preference ranks.

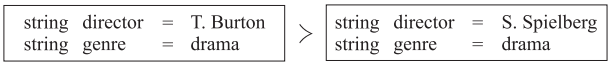


Figure 2.4: Priority condition example.

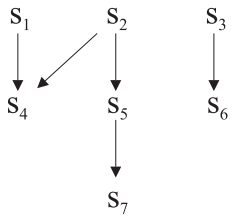
Priority Conditions	Graph	Preference Ranks
$s_1 \succ s_4$ $s_2 \succ s_4$ $s_2 \succ s_5$ $s_3 \succ s_6$ $s_5 \succ s_7$ $s_2 \succ s_7$		s_1, s_2, s_3: 1 s_4, s_5, s_6: 0.5 s_7: 0.33

Figure 2.5: Extracting preference ranks.

Summarizing, subscriptions are augmented with preference ranks specified using either the quantitative or the qualitative approach. A preference rank indicates the user's degree of interest for the corresponding subscription. The higher the preference rank, the more important the events covered by the corresponding subscription.

2.3 Computing Event Ranks

Let P^X be the set of preferential subscriptions of user X. We use these preferential subscriptions to rank the published events and deliver to X only the highest ranked ones. We define the *rank* of an event to be a function $\mathcal{F}$ of the preference ranks of the user subscriptions that cover it:

Definition 2.6 (Event Rank)**.** Assume an event e, a user X and the set P^X of the user's preferential subscriptions. Assume further the set $P_e^X = \{(s_1, prefrank_1^X), \ldots, (s_m, prefrank_m^X)\}$, $P_e^X \subseteq P^X$, for which s_i covers e, $1 \leq i \leq m$. The event rank of e for X is equal to $rank(e, X) = \mathcal{F}(prefrank_1^X, \ldots, prefrank_m^X)$, where $\mathcal{F}$ is a monotonic function.

An event e_1 is more preferable for user X to an event e_2, if and only if, it has a higher event rank for X than e_2.

As the aggregation function $\mathcal{F}$ for computing the rank of an event, we may use the maximum, mean, minimum or a weighted sum of the preference ranks of its covering subscriptions. Furthermore, instead of using the preference ranks of all covering subscriptions, we can use only those of the *most specific* ones. For example, assume the event of Figure 2.2a and the preferential subscriptions ($\{genre = drama\}$, 0.9) and ($\{genre = drama, director = T.\ Burton\}$, 0.8) (for ease of presentation, we omit the type of each attribute). Both subscriptions cover the event. Between the two, the latter subscription is more specific than the former one, in the sense that in the latter subscription the user

poses an additional, more specific requirement to movies. Thus, intuitively, the preference rank of the latter subscription should superimpose that of the former one, whenever an event matches both of them. Formally, a subscription $s \in P_e^X$ is a most specific one if no other subscription in P_e^X is covered by it. The cover relation between two subscriptions is defined as follows:

Definition 2.7 (Cover Between Subscriptions)**.** Given two subscriptions s_i and s_j, s_i covers s_j, if and only if, for each event e such that s_j covers e, s_i covers e.

In general, computing ranks may increase the complexity of the process of matching events with subscriptions. In traditional publish/subscribe systems, for matching to be completed successfully, it suffices to find just one subscription that covers the event, whereas for computing the rank of an event, we may need to locate all covering subscriptions.

Chapter 3

Timing Policies

Having defined event ranks, our goal is to send notifications only for the top-k events to each user, i.e. the events with the k highest ranks. Since events are continuously published and matched with subscriptions, we need to specify the timing period over which the top-k events are computed. In this chapter, we examine the various timing policies that can be applied to a publish/subscribe system, since different timing policies can affect whether an event belongs to a user's top-k results or not. In particular, we examine the following timing policies: (i) *continuous*, in which events are forwarded to interested subscribers at the time of their publication, (ii) *periodic*, in which events are forwarded at predefined time intervals and (iii) *sliding window*, which combines the previous policies.

3.1 Continuous Timing Policy

In the continuous timing policy, a newly published event is delivered to a user, if and only if, it matches one of the user's subscriptions and the user has not already received k events with higher ranks than the new one's. Since new events are constantly produced, it is possible for very old but highly preferable events to prevent any new ones from reaching the user. For instance, consider the example in Figure 3.1. For simplicity, we assume a single user, say John, who has defined a number of preferential subscriptions for movies. Assume that a movie theater publishes the events $e_1, e_2, \ldots, e_6$ in that order, at the time shown on the left of each event, and that John is interested in the top-2 results. Assume

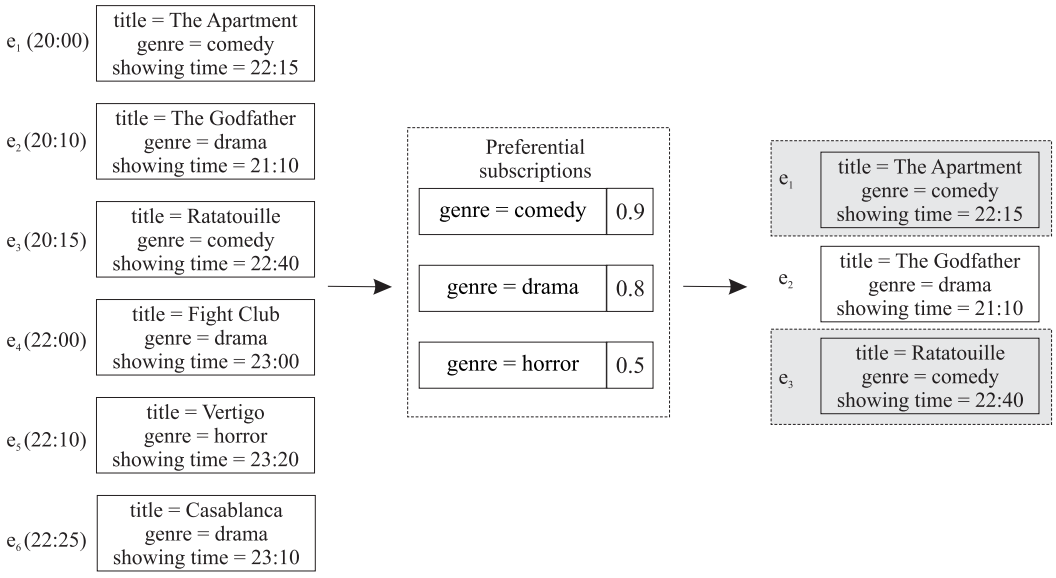


Figure 3.1: Continuous (no expiration): top-2 events for John at 22:55.

further that as an aggregation function $\mathcal{F}$ we use the maximum value of the preference ranks of the covering subscriptions, i.e. $\mathcal{F} = \max$. e_1 and e_2 will be delivered to John, since they are the first two events that are covered by his subscriptions. e_3 is equally preferred to e_1 and will therefore also be delivered to John. Since e_4, e_5 and e_6 are less preferable to the current top-2 results, none of them will be delivered. If John checks his top-2 results at 22:55, he will only find movies that he can no longer watch (the top-2 results at 22:55 are marked with gray color in the figure), even though some events about interesting movies that start at 23:00 have been published.

To overcome this problem, we can associate with each published event e an expiration time $e.exp$. The event is considered *valid* only while $e.exp$ has not expired. This way, older events which have expired do not prevent new ones from reaching the users. An alternative way to set the expiration time for an event is by letting the user define a *refresh time* along with each subscription. Then, the expiration time of an event e covered by a user subscription s with refresh time r is set to $t+r$, where t is the time that e is matched with s. Note that in this approach, a specific event does not have a single expiration time but instead, it is associated with a different one for each user.

Next, we define the conditions under which a published event belongs to the top-k results of a user using expiration times.

Definition 3.1 (Continuous Top-k)**.** Let P^X be the set of preferential subscriptions of user X. An event e published at time t_e belongs to the top-k events of X, if and only if, e is covered by at least one subscription s of a preferential subscription $ps \in P^X$ and there are no k events $e_1, \ldots, e_k$ with $e_i.exp > t_e$ and $rank(e_i, X) > rank(e, X)$, $1 \leq i \leq k$, that have already been forwarded to X.

In the previous example, assume now that each event expires at the showing time of the corresponding movie (see Figure 3.2). e_1, e_2 and e_3 will be delivered to the user

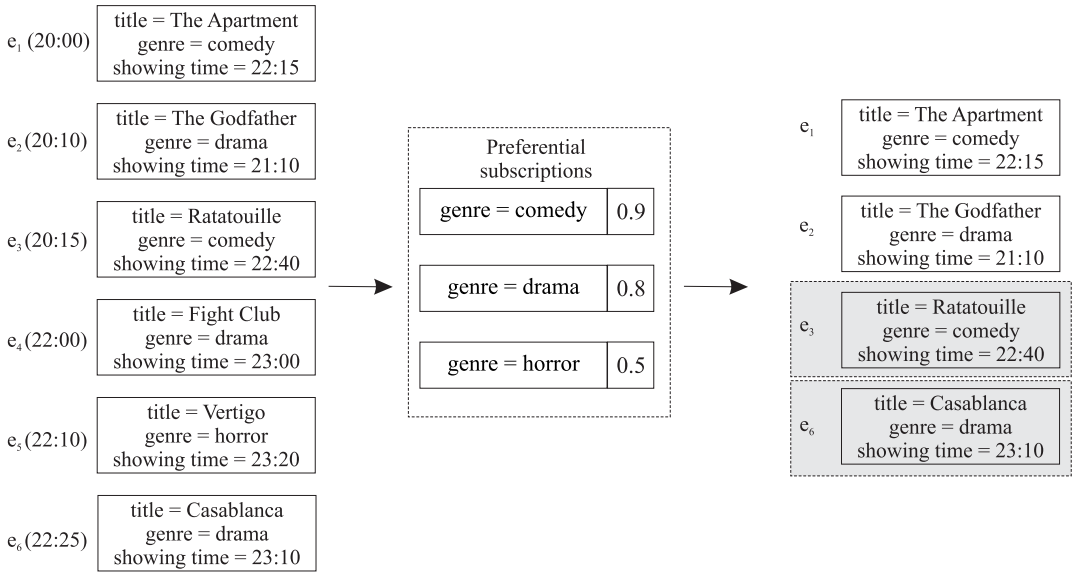


Figure 3.2: Continuous (with expiration): top-2 events for John at 22:55.

as before. By the time e_4 and e_5 are published, the top-2 events have not expired and therefore, e_4 and e_5 will not be delivered to the user. Finally, e_6 will be delivered to John, since at the time of its publication e_1 has expired. Therefore, when John checks his top-2 results at 22:55, he will find an interesting movie to watch ("Casablanca").

When the continuous timing policy is used, the number of delivered events depends on the relative order of their publication. For example, if events are produced in ascending order with regard to their rank, all of them will be delivered, while if they are produced in descending order only a portion of them will.

3.2 Periodic Timing Policy

An alternative timing policy is the periodic one. In this approach, time is divided into periods of duration T and top-k events are computed within each period. As with the continuous timing policy, events can be associated with expiration times, so that only valid events are delivered to users. For instance, considering the previous example, for a time period that begins at 20:00 and ends at 20:30, the top-2 results are the events e_1 and e_2, while from 20:00 to 22.20, the top-2 results are the events e_3 and e_4, since e_1 and e_2 have already expired, e_5 is less preferable to e_4 and e_6 has not been published yet. Clearly, in the periodic timing policy, the computation of the top-k results depends on the duration of the period.

Definition 3.2 (Periodic Top-k)**.** Let P^X be the set of preferential subscriptions of user X. An event e published at time t_e in the current period belongs to the top-k events of the period for X, if and only if, e is covered by at least one subscription s appearing in a preferential subscription $ps \in P^X$, $e.exp > \lceil t_e/T \rceil \cdot T$ and there are no k events $e_1, \ldots, e_k$

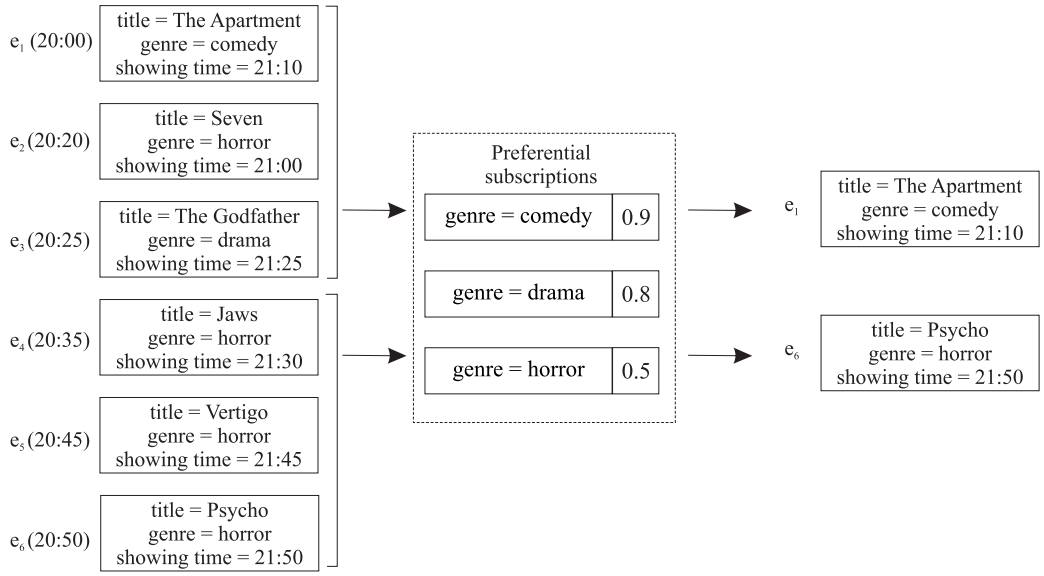


Figure 3.3: Periodic: top-1 event for $T = 30$ min.

published in the period with $e_i.exp > \lceil t_e/T \rceil \cdot T$ and $rank(e_i, X) > rank(e, X)$, $1 \leq i \leq k$.

For any time interval of length c, the number of events that eventually reach the users is bounded by k and equal to $k \cdot \lfloor c/T \rfloor$. The order of events published within a specific period does not affect the top-k results for it.

3.3 Sliding Window Timing Policy

In the periodic timing policy, the top-k computation starts anew in the beginning of each period. Therefore, the ranks of events received by the user may end up being rather arbitrary. For example, high-ranked events appearing in periods with many other high-ranked ones may not be delivered to the user, whereas low-ranked publications appearing in periods with a small number of high-ranked ones may be delivered. For instance, assume the events e_1, e_2, ..., e_6 of Figure 3.3 and John's previous subscriptions. Assume now that John is interested in the top-1 result. For the time period from 20:00 to 20:30 the best result is e_1, while for the period from 20:30 to 21:00 the best result is e_6, which means that e_3 is not delivered to John, even though it is more preferable than e_6. Note that we select to resolve ties among events by picking the most recent one to improve the freshness of results.

To overcome this issue, we use sliding event-windows and start the computation anew at each new matching event. That is, we compute the top-k events for a user based only on the events published during an event-window of length w, i.e. based only on the w most recent matching events. For example, assume a window of length $w = 3$ and the previous published events (see Figure 3.4). We use the notation W_i to refer to the events that are included in the i^{th} window. As depicted in the figure, if we are interested in the

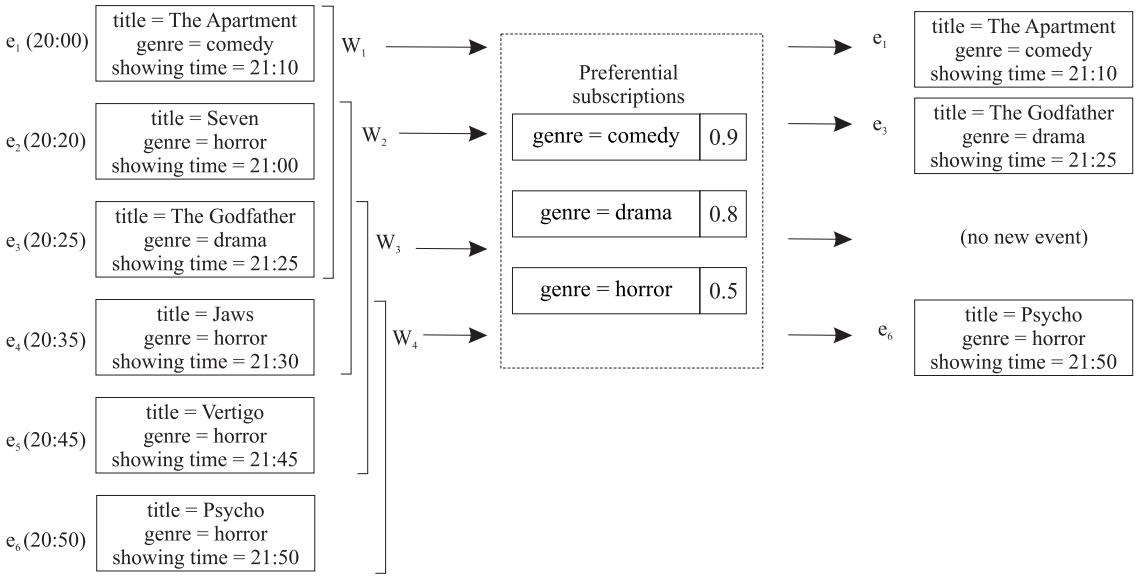


Figure 3.4: Sliding window: top-1 event for $w = 3$.

top-1 result, the first window W_1 returns the event e_1, W_2 returns e_3, W_3 returns no event and so on.

Observe that between two consequent event-windows, at most one event is delivered. To see this, assume a window W_1 and its following window W_2, both of length w, and the two sets A_1, A_2 with the top-k events for W_1 and W_2 respectively. Since W_1 and W_2 have $w-1$ common events, let $W_1 = (e_1, e_2, \ldots, e_w)$ and $W_2 = (e_2, e_3, \ldots, e_{w+1})$. When e_{w+1} is published, one of the following holds:

- $e_1 \in A_1$. Then $A_2 = (A_1 - \{e_1\}) \cup \{e'\}$, where e' is either e_{w+1} or e' is an event that was published in W_1 for which $e' \notin A_1$ holds, or
- $e_1 \notin A_1$. Then $A_2 = A_1$ or $A_2 = (A_1 - \{e'\}) \cup \{e_{w+1}\}$, where e' is an event that was published in W_1.

In any case, at most one event enters the set A_2, thus:

Property 3.1. Between two consequent event-windows, when events do not expire, at most one new event is delivered to the user.

As in the continuous timing policy, the number of delivered events depends on their publication order with regards to their relative ranks. In this policy as well, an event e may be associated with a specific expiration time $e.exp$. The definition of sliding window top-k is similar to the periodic one.

Summarizing, the timing policies determine the range of events over which the top-k computation is performed. In the continuous timing policy, the top-k events are selected over all previously published events, while in the periodic and sliding window timing policies, they are selected over the events published in the current period or window respectively.

3.4 Event Delivery

Independently of how the top-k events are computed by the system, we offer to users two ways to view their results: proactive and on-demand. In the *proactive* approach, top-ranked results are computed by the system, using any one of the three timing policies, and delivered to the users at specific moments in time. Top-results are gathered by the event-notification service and forwarded to users every T' time units. Note that, even though the use of the periodic timing policy with length T to compute top-ranked events is allowed in conjunction with proactive delivery, T is not necessarily equal to T'. In this case, whenever new results are to be delivered to the user, the forwarded results are those of the last completed period.

In the *on-demand* approach, top-results are gathered again by the event-notification service and are forwarded to the user upon request. All three timing policies can be used with this method as well. However, the one intuitively most suitable for it is that of the sliding window timing policy, since in this case, the user receives the most recent to the time of the request top-ranked events.

Chapter 4

Events Diversity

In a typical publish/subscribe system, many events are continuously published and competing for a position in a user's top-k results. Many times, the results that eventually reach the user are very similar to each other. However, it is often desirable that these results exhibit some diversity. In this chapter, we examine how to arrange the results delivered to users in order to reduce their similarity. First, we present a technique for locating k diverse results and then focus on how to compute the top-k results based on both their diversity and their rank.

4.1 Events Distance

Instead of overwhelming users with published events that are all very similar to each other, we opt to select a representative set of events according to their diversity. To measure the *diversity* of events, i.e. how different they are, we first define the distance between two events. Without loss of generality, we assume that the events have the same number of attributes. Otherwise, we can append a sufficient number of "dummy" attributes to the one with the smaller number of attributes.

Definition 4.1 (Event Distance)**.** Given two events $e_1 = \{a_1, \ldots, a_p\}$ and $e_2 = \{a'_1, \ldots, a'_p\}$, the distance between e_1 and e_2 is:

$$DIS(e_1, e_2) = 1 - \frac{\sum_{i=1}^{p} w_i \delta_i}{\sum_{i=1}^{p} w_i}, \qquad \text{where } \delta_i = \begin{cases} 1 & \text{if } a_i = a'_i \\ 0 & \text{otherwise} \end{cases}$$

and each w_i is an attribute specific weight, $1 \leq i \leq p$.

According to the above definition, given two events, their distance decreases as the number of their common attributes increases. The weight assigned to each attribute is application dependent, since for some applications, some attributes may be more influential than others. In the lack of application-specific information, we can assign equal weights to all attributes.

Given the set M of all matching events for a user, we would like to deliver to the user a list L, $L \subseteq M$, with the k most diverse ones. To measure the diversity of the events that belong to a list, we use the *List Diversity* that computes the average distance of all pairs of events in L.

Definition 4.2 (List Diversity)**.** Given a list of m events $L = (e_1, \ldots, e_m)$, the list diversity of L is:

$$DIV(L) = \frac{\sum_{i=1}^{m} \sum_{j=i+1}^{m} DIS(e_i, e_j)}{(m-1)m/2}$$

A brute-force method to identify the k most diverse events from M is to first produce all $\binom{q}{k}$ possible combinations of k events, where q is the number of elements in M, and then pick the one with the maximum list diversity. The complexity of this process in terms of the required event distance operations is equal to $\frac{q!}{k!(q-k)!} \cdot \frac{(k-1)k}{2}$.

To reduce the complexity, we use the following heuristic. We incrementally construct a diverse subset of events based on the *Event-List Distance*, defined as follows:

Definition 4.3 (Event-List Distance)**.** Given an event e and a list of m events $L = (e_1, \ldots, e_m)$, the event-list distance between e and L is:

$$DIS(e, L) = \frac{\sum_{i=1}^{m} DIS(e, e_i)}{m}$$

Given the set $M = \{e_1, \ldots, e_q\}$ of q events, $q > k$, our goal is to produce a list $L = (e'_1, \ldots, e'_k)$, $e'_i \in M$, $1 \leq i \leq k$, with the k most diverse events. Initially, we consider an empty list L. We first add to L a random event of M. Then, we incrementally construct L by selecting events of M according to their distance from events previously inserted into L. In particular, we compute the distances $DIS(e_i, L)$, $\forall e_i$, such that $e_i \in M$ and $e_i \notin L$ and add to L the one with the maximum distance. This process is repeated until k events have been added to the list. Following this method, the required event distance operations are equal to $(q-1) + 2(q-2) + \ldots + (k-1)(q-k+1)$.

To compare the quality of our heuristic against the brute-force method, we perform the following experiment. First, we create sets of q random events. For simplicity reasons, each event consists of a number in [0, 100] and their distance is measured as the absolute value of the difference of their values. Then, we select k out of them (i) randomly, (ii) using our heuristic and (iii) using the brute-force method. In Table 4.1, we show the average List-Diversity for the lists produced in each case. We see that the heuristic performs much better than the random approach and produces lists with List-Diversity slightly smaller than the brute-force method.

Table 4.1: List-Diversity for the random, heuristic and brute-force methods.

q	k	Random	Heuristic	Brute-Force
20	2	35.76	71.88	90.45
	4	193.23	330.69	349.27
	8	938.72	1302.50	1314.64
40	2	34.14	74.86	95.86
	4	200.35	353.78	375.84
	8	944.63	1427.09	1447.93

4.2 Diverse Top-k Events

The method described above delivers to the user a set of diverse events but ignores their importance according to the preferential subscriptions. However, we would also like to consider the user preferences when selecting the events to be forwarded.

Therefore, to compute the top-k diverse events for a user X, i.e. a list L^X, we modify the above method to also take into account the ranks of the events. Initially, instead of selecting the first event to be added to L^X randomly, we pick the one with the highest rank, since this is the most preferable to the user. When selecting the next event to be added to L^X, we choose the one with the maximum value of *divrank*, where *divrank* is a function that takes as input an event e and a user X and combines the rank of e for X and its distance from the already selected events for that user using the formula:

$$divrank(e, X) = \sigma \cdot rank(e, X) + (1 - \sigma) \cdot DIS(e, L^X)$$

where $\sigma \in [0, 1]$. When $\sigma = 0$ (respectively $\sigma = 1$), only the distance (respectively rank) is taken into account.

In the following example, we apply the above procedure to six events e_1, e_2 ..., e_6. To simplify our example, we assume that all events have only the attribute "genre" with values equal to "comedy", "drama", "drama", "drama", "horror", "sci-fi" and ranks 0.9, 0.75, 0.75, 0.75, 0.65, 0.6 respectively. Figure 4.1 shows the trace of the method applied on our example for $\sigma = 0.5$.

Let L be the current list of diverse top-k events. We observe that after the insertion of an event e to L, $L' = L.append(e)$, the distances of all other events that have not yet entered the diverse top-k events from L' are affected only by the presence of e. Therefore, we can reduce the number of performed operations of the above procedure by exploiting the following property:

Property 4.1. Given an event e and two lists L and $L' = L.append(e)$, the distance of an event e' from L' is:

$$DIS(e', L') = \frac{DIS(e', L) \cdot (|L| - 1) + DIS(e, e')}{|L|}$$

Ranking of events						
$e_1(comedy), e_2(drama), e_3(drama), e_4(drama), e_5(horror), e_6(sci\text{-}fi)$						
Top-4 events based on their ranks						
(e_1, e_2, e_3, e_4)						
Diverse Top-4 events						
$DIS(e_1, L)$	$DIS(e_2, L)$	$DIS(e_3, L)$	$DIS(e_4, L)$	$DIS(e_5, L)$	$DIS(e_5, L)$	$L = \emptyset$
-	-	-	-	-	-	$L = (e_1)$
-	0.875	0.875	0.875	0.825	0.800	$L = (e_1, e_2)$
-	-	0.625	0.625	0.825	0.800	$L = (e_1, e_2, e_5)$
-	-	0.708	0.708	-	0.800	$L = (e_1, e_2, e_5, e_6)$

Figure 4.1: Computing top-4 diverse events.

Algorithm 1 (*Diverse Top-k Events Algorithm*) summarizes the above procedure.

Input: A set M of matching events for user X.
Output: A list L of diverse top-k events.

1: **begin**
2: $L = \emptyset$;
3: find the event $e \in M$ with the maximum $rank(e, X)$;
4: $L = L.append(e)$;
5: $M = M.remove(e)$;
6: **for all** events $e_i \in M$ **do**
7: $distance_{e_i} = 0$;
8: **while** $|L| < k$ **do**
9: **for all** events $e_i \in M$ **do**
10: $distance_{e_i} = \frac{distance_{e_i} \cdot (|L|-1) + DIS(e_i, L.last)}{|L|}$;
11: find the event e with the maximum
$divrank(e, X) = \sigma \cdot rank(e, X) + (1 - \sigma) \cdot distance_e$;
12: $L = L.append(e)$;
13: $M = M.remove(e)$;
14: **return** L;
15: **end**

Algorithm 1: Diverse Top-k Events Algorithm.

Note that diversification may increase the number of events delivered to the user. For example, in the sliding window timing policy, when diversifying the top-k results, we may deliver up to k events at each event-window, that is, in this case, Property 3.1 does not hold. To illustrate this, let W_1, W_2 be two consequent event-windows and e_1, e_2, e_3, e_4 a number of events. Assume that $W_1 = (e_1, e_2, e_3)$, $W_2 = (e_2, e_3, e_4)$ and also that e_1 is a horror movie directed by T. Burton with rank 0.9, e_2 is an A. Hitscock's horror movie with rank 0.85, e_3 is a S. Spielberg's drama movie with rank 0.8 and finally, e_4 is a Q. Tarantino's drama movie with rank 0.9. Applying Algorithm 1, the top-2 events for

W_1 are the events e_1 and e_3, while the top-2 events for W_2 are e_4 and e_2, which means that more than one event will be delivered in W_2.

In conclusion, to compute the final rank of an event, we use the *divrank* function that combines both preferences and diversity.

Chapter 5

Ranking in Publish/Subscribe

In this chapter, we outline a method for matching events with subscriptions and for computing event ranks. To this end, we introduce the *preferential subscription graph* for organizing our preferential subscriptions. We also show how to compute the top-k results for each timing policy.

5.1 Preferential Subscription Graph

To reduce the complexity of the matching process between events and subscriptions, it is useful to organize the subscriptions using a graph. We use preferential subscriptions to construct a directed acyclic graph, called *preferential subscription graph*, or PSG. In a preferential subscription graph, nodes correspond to subscriptions and edges to cover relations between subscriptions (see Definition 2.7). Let P be the set of all preferential subscriptions, i.e. the preferential subscriptions defined by all users. For each subscription $s_i \in S_P$, where S_P is the set of all subscriptions in P, we maintain a set of pairs, called *PrefRank Set*, of the form $(X,\ prefrank_i^X)$, where X is a user and $prefrank_i^X$ is the preference rank of X for s_i. A subscription s_i is associated with the pair $(X,\ prefrank_i^X)$, if and only if, a preferential subscription $ps_i^X = (s_i, prefrank_i^X)$ exists in P. Given a set of users $\mathcal{U}$, for each $s_i \in S_P$, the PrefRank Set is the set $R_i = \{(X,\ prefrank_i^X) \mid (s_i, prefrank_i^X) \in P,\ X \in \mathcal{U}\}$. Formally:

Definition 5.1 (Preferential Subscription Graph)**.** Let P be a set of preferential subscriptions and S_P the set of all subscriptions in P. A Preferential Subscription Graph

Subscription	John's prefrank	Anna's prefrank
$\{cinema = ster\}$	0.5	-
$\{genre = drama, time > 21{:}00\}$	0.7	-
$\{genre = comedy\}$	0.7	-
$\{cinema = ster, time < 20{:}00\}$	0.4	-
$\{cinema = ster, genre = drama, time > 21{:}00\}$	0.9	0.6
$\{cinema = odeon, genre = drama, time > 21{:}00\}$	0.3	0.9
$\{genre = comedy, time > 23{:}00\}$	-	0.8
$\{cinema = ster, genre = drama, time > 23{:}00\}$	-	0.8

Figure 5.1: John and Anna's preferential subscriptions.

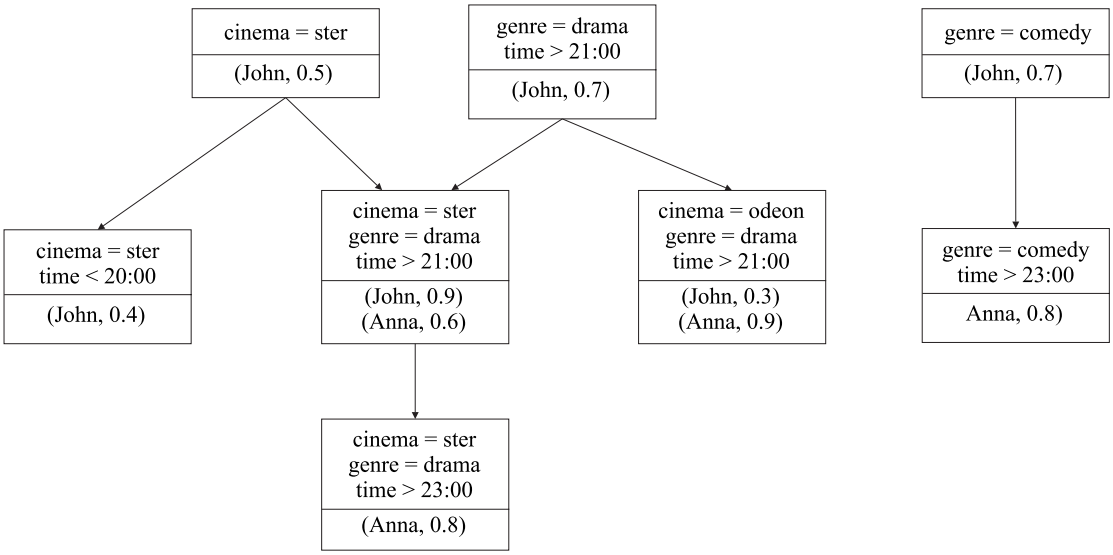


Figure 5.2: Preferential subscription graph example.

$PSG_P(V_P, E_P)$ is a directed acyclic graph, where for each different $s_i \in S_P$, there exists a node v_i, $v_i \in V_P$, of the form (s_i, R_i), where R_i is the PrefRank Set of s_i. Given two nodes v_i, v_j, there exists an edge from v_i to v_j, $(v_i, v_j) \in E_P$, if and only if, s_i covers s_j and there is no node v'_j such that s_i covers s'_j and s'_j covers s_j.

For example, assume two users, John and Anna, who express the preferential subscriptions of Figure 5.1. For those preferential subscriptions, the graph of Figure 5.2 is constructed.

The preferential subscription graph resembles the *filters poset* data structure proposed in [7]. Whereas the filters poset represents a partially ordered set of subscriptions, the preferential subscription graph is based on subscriptions enhanced with preferences. Next, we will describe how preferential subscription graphs are constructed.

The partial order of the subscriptions of a preferential subscription graph PSG is defined by the covering relation between subscriptions. We say that a subscription s_1 is an *immediate predecessor* of a subscription s_2, if and only if, s_1 covers s_2 and there is no other subscription s_3 in PSG such that s_1 covers s_3 and s_3 covers s_2. s_2 is called

an *immediate successor* of s_1. Subscriptions with no predecessors in the graph are called *roots*. For every new preferential subscription $ps_i^X = (s_i, prefrank_i^X)$, one of the following holds:

- s_i already exists as a node in the graph
- s_i is added as a root node
- s_i is inserted somewhere in the graph with a nonempty set of predecessors

Upon receiving a preferential subscription $ps_i^X = (s_i, prefrank_i^X)$, the graph is traversed in a breadth-first way and two (possibly empty) sets are identified, namely $\overline{s_i}$ and $\underline{s_i}$, representing the immediate predecessors and the immediate successors of s_i respectively. If $\overline{s_i} = \underline{s_i} = \{s_i\}$, then s_i already exists in the graph and the pair $(X, prefrank_i^X)$ is simply added to the subscription's PrefRank Set. If X had issued the same subscription in the past associated with a different preference rank, then the corresponding value is updated instead. If s_i does not already exist in the graph, it is inserted between $\overline{s_i}$ and $\underline{s_i}$ and its PrefRank Set is initialized appropriately. In the special case when $\overline{s_i} = \emptyset$, s_i is inserted as a root node. Upon receiving an unsubcription request, any pairs containing X are removed from all the PrefRank Sets of the subscriptions defined in the request. Any subscriptions left with an empty PrefRank Set are removed as well.

5.2 Forwarding Events

To show how the top-k results for each user are computed, we first assume a single server maintaining a preferential subscription graph PSG. This server acts as an access point for all subscribers and publishers. Next, we describe how event delivery is performed for each of the three timing policies, considering first using ranks based solely on preferences.

Typically, publish/subscribe systems are stateless, in that, they do not maintain any information about previously delivered events. However, to provide users with the top-ranked matching events, we may need to maintain some information about previously delivered events as well as buffer some published events prior to their delivery or dismissal.

In the continuous timing policy, we need to maintain information about previously forwarded top-ranked events. Specifically, the server maintains a list of k elements for each of the subscribers that are connected to it. These lists contain elements of the form $(rank, expiration)$, where $rank$ is a numeric value and $expiration$ is a time field. The $rank$ part of such a pair represents the rank of an event that has already been forwarded to the corresponding user and expires at time $expiration$. Only the ranks corresponding to the top-k most preferable valid events that have been already sent to the users appear in these lists.

All lists are initially empty. Whenever the server receives an event e, it walks through its PSG to find all subscriptions that cover e. For each subscriber X associated with

Input: An event e and a preferential subscription graph PSG.
Output: The set of subscribers $ResSet$ that e will be forwarded to.

1: **begin**
2: $ResSet = \emptyset$;
3: $tmpR = \emptyset$; /* temporary PrefRank Set */
4: **for all** nodes v_i in PSG **do**
5: **if** s_i covers e **then**
6: $tmpR = tmpR \cup R_i$;
7: **for all** subscribers X that appear in $tmpR$ **do**
8: $rank(e, X) = max\{prefrank_1^X, \ldots, prefrank_{m^X}^X\}$, where $(X, prefrank_i^X) \in tmpR$, $1 \leq i \leq m^X$;
9: **for all** elements i in $list^X$ **do**
10: **if** i has expired **then**
11: remove i from $list^X$;
12: **if** $list^X$ contains less than k elements **then**
13: add $(rank(e, X), e.exp)$ to $list^X$;
14: $ResSet = ResSet \cup j$;
15: **else**
16: find the element i of $list^X$ with the minimum rank;
17: **if** $rank(e, X) > i.rank$ **then**
18: remove i from $list^X$;
19: add $(rank(e, X), e.exp)$ to $list^X$;
20: $ResSet = ResSet \cup j$;
21: **return** $ResSet$;
22: **end**

Algorithm 2: Continuous Forwarding Events Algorithm

at least one of these subscriptions, an event rank $rank(e, X)$ is computed. In this work, we assume that the preference ranks associated with the various subscriptions are indicators of positive interest, thus, we use as an aggregation function $\mathcal{F}$ the maximum value of the preference ranks of the covering subscriptions. Assuming that m subscriptions $s_1, s_2, \ldots, s_m$ submitted by X cover e, $rank(e, X) = max\ \{prefrank_1^X, prefrank_2^X, \ldots, prefrank_m^X\}$.

After that, the corresponding list with the k element pairs, denoted $list^X$, is checked and all elements which have expired are removed. If $list^X$ contains now less than k elements, e is forwarded to X and the pair $(rank(e, X), e.exp)$ is added to the list, where $e.exp$ is the expiration time of e. Otherwise, e is forwarded to X only if $rank(e, X)$ is greater or equal to the rank of the element with the minimum rank in the list. In this case, this element is replaced by $(rank(e, X), e.exp)$. Note that, a more recent event equally important to an older one is forwarded to the user to favor fresh data over equally-ranked old ones. The process described above is summarized in the *Continuous Forwarding Events Algorithm* shown in Algorithm 2.

Next, we show the completeness and correctness of Algorithm 2. First, we will show that if an event e belongs to the top-k results of user X, then it will be forwarded to X. Assume for the purpose of contradiction, that such an event is not forwarded to X. Let $rank(e, X)$ be the rank of e for X. Since e is not forwarded to X, there exist k valid events $e_1, \ldots, e_k$ with ranks $rank(e_1, X), \ldots, rank(e_k, X)$ such that $rank(e_i, X) > rank(e, X)$, $1 \leq i \leq k$. This means that e does not belong to the top-k results of user X, which violates our assumption. Next, we proceed with showing that if an event e is forwarded to X, then it belongs to the user's top-k results. For the purpose of contradiction, assume that e does not belong to the user's top-k results. This means that there exist k valid events $e_1, \ldots, e_k$ with ranks $rank(e_1, X), \ldots, rank(e_k, X)$ such that $rank(e_i, X) > rank(n, X)$, $1 \leq i \leq k$. Therefore, according to Algorithm 2 (line 21), e will not be forwarded to X, which is a contradiction.

Note that it is not necessary to walk through all nodes of the preferential subscription graph to locate the subscriptions that cover a specific event e. We may safely ignore a node v with subscription s for which there is no other node v' with subscription s', such that s' covers s and at the same time s' covers e. This way, entire paths of the graph can be pruned and not used in the matching process. For example, in Figure 5.2, if an incoming event is not covered by $\{cinema = ster\}$, then it is certainly not covered by $\{cinema = ster,\ time < 20{:}00\}$, $\{cinema = ster,\ genre = drama,\ time > 21{:}00\}$ or $\{cinema = ster,\ genre = drama,\ time > 23{:}00\}$ either and therefore, those subscriptions do not have to be checked against the event.

In the case of the periodic timing policy, there is no need to maintain information about previously sent events. Instead, the server buffers all events published during the current period. At the end of the period, we compute the ranks of the buffered events for all users and deliver the corresponding top-k events to each of them. Only events that are still valid, i.e. not expired, at the end of the period are considered.

The computation of top-k results in the sliding window timing policy is similar to the periodic one. However, in this case, the server needs to buffer the w most recent matching events for each subscriber. Whenever a new event is added to a buffer, the top-k events of the buffer are computed. The ones that have not already been forwarded to the user in the past are delivered (events may have been forwarded in the past because consequent windows overlap). In the case that expiration times are used, events are removed from the buffer upon expiration.

Since in the continuous timing policy we do not maintain any information about the content of previously forwarded events but only about their ranks, we opt not to apply the diversity technique. The focus of this timing policy is the fast delivery of top-ranked events. The diversity technique is best suited for the periodic and sliding window policies. Concerning the periodic policy, at the end of each period, we apply the Diverse Top-k Events Algorithm (Algorithm 1) to compute the top-k diverse events for each user. Similarly, in the sliding window policy, whenever a new event is added to a buffer, its contents are re-arranged according to the events' *divranks*. After this re-arrangement

takes place, the events that belong to the top-k ones in the buffer are forwarded to the user, unless they have already been forwarded in the past.

5.3 Topology of Servers

An event-notification service can be implemented over various architectures. At one extreme, a centralized approach can be followed, e.g. [13]. In this case, a single server gathers all subscriptions and notifications and carries out the matching process. However, due to the nature of such systems, where participants are physically distributed across the globe, a distributed architecture is more scalable. When more than one server exists in the network, each server runs Algorithm 2 for its own preferential subscription graph. Events are propagated among servers based on the server topology. The servers of the system are responsible for collecting all the published events and carrying out the selection process, i.e. delivering each event only to the subscribers that have declared their interest to it.

In this work, we consider a hierarchical topology, where the servers that implement the event-notification service are connected to each other to form a hierarchy. Each publisher and subscriber is connected to one of the servers in the hierarchy. We wish to organize the participants of the network in an efficient way, i.e. in a way that will reduce the number of messages exchanged between the servers and the complexity of the maintained data structures. One way to achieve this is by placing subscribers with similar subscriptions nearby in the hierarchy, so that the events covered by those subscriptions need to be propagated only toward this part of the hierarchy. To do this, we exploit the structure of preferential subscription graphs and more specifically, the covering relations between their nodes. We observe that all events that are covered by some subscription of a preferential subscription graph PSG are also covered by at least one of the most generic subscriptions in it. Therefore, only events covered by the subscriptions in nodes with no incoming edges, i.e. root nodes, need to be propagated to the server maintaining the PSG. So, in order to reduce the exchanged messages we have to minimize the number of root nodes in the various preferential subscription graphs of the system.

While in most publish/subscribe systems new subscribers randomly select a server to connect to, in our approach, when a new subscriber enters the network, it probes a number of servers and chooses one of them according to the following criteria:

- (*Criterion 1*) The number of new root nodes added to the server's preferential subscription graph. The smaller the number of such nodes, the fewer the additional events that should be propagated to the server in the future.
- (*Criterion 2*) The number of nodes in the server's preferential subscription graph. The fewer the nodes in the graph, the lower the complexity of searching it.
- (*Criterion 3*) The number of existing subscriptions in the graph covered by the new

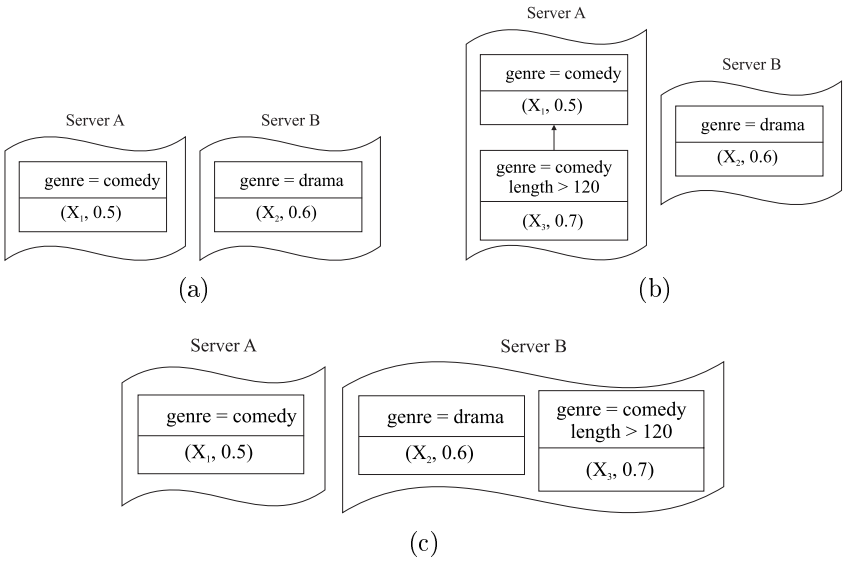


Figure 5.3: Clustering.

ones. The closer to the root level of the graph the new subscriptions will be inserted, the quicker future pruning will occur.

A new subscriber may first use *Criterion 1*, and in case of a tie, *Criterion 2* and/or *Criterion 3* to select a server. For example, consider the case of Figure 5.3a where there are two servers, Server A and Server B, both already storing some user subscriptions from subscribers X_1 and X_2. Assume that a new subscriber X_3 wishes to insert a new preferential subscription ($\{genre = comedy, length > 120\}$, 0.7) to the system. If X_3 chooses Server A to subscribe, the result will be the one shown in Figure 5.3b. If X_3 chooses Server B, the result will be the one shown in Figure 5.3c. Using the first criterion, X_3 will choose to join Server A because in this case no new root nodes will be added to the preferential subscription graph of Server A and thus, no new message traffic will be generated (except from the messages sent from Server A to X_3). This clustering method is non-preemptive, in the sense that it does not modify the existing underlying structures of the system.

Chapter 6

Evaluation

To evaluate our approach, we have extended the SIENA event notification service [4], a multi-threaded publish/subscribe implementation, to include preferential subscriptions and perform ranked delivery of diverse events according to the three timing policies. We refer to our prototype as PrefSIENA [3]. In this chapter, we first describe the dataset used to evaluate our approach and then present our experimental results.

6.1 Dataset

To evaluate the performance of our model, we use a real dataset, available online at [2]. The dataset consists of data derived from the Internet Movie Database (IMDB) [1], a major database of information about movies, actors, film makers etc., online since 1990. The dataset contains information about 58788 movies. More specifically, for each movie the information of Table 6.1 is available.

string	title	=	Big Fish
integer	year	=	2004
integer	length	=	125
double	rating	=	9.0
integer	votes	=	180
string	mpaa	=	PG-13
string	genre	=	drama

(a) Generated event.

string	genre	=	drama	0.8
integer	length	>	100	
integer	year	<	1980	

string	genre	=	romance	0.6
string	year	>	1990	
string	mpaa	=	PG-13	

(b) Generated preferential subscriptions.

Figure 6.1: Generated data.

Table 6.1: Movies dataset properties.

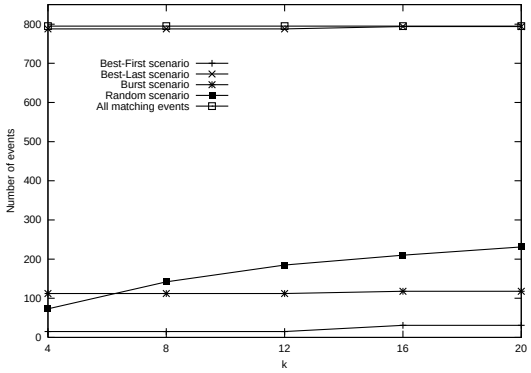
Attribute	Description	Type
title	The title of the movie	string
year	The year of release	integer
budget	The budget of the movie (in US dollars)	double
length	The length of the movie (in minutes)	integer
rating	The average IMDB user rating for the movie	double
votes	The number of IMDB users who rated the movie	integer
r1-r10	The distributions of votes for each rating (from 1.0 to 10.0)	double
mpaa	The Motion Picture Association of America (MPAA) rating for the movie	string
genre	The genre of the movie (possibly more than one)	string

Data generation: Publishers generate events as follows. At random intervals, each publisher uniformly selects m_P numbers from 1 to 58788. For each of the corresponding m_P movies the publisher creates a new event consisting of the title, year, length, rating, votes, mpaa rating and the genre(s) of the movie. An example of such an event can be seen in Figure 6.1a.

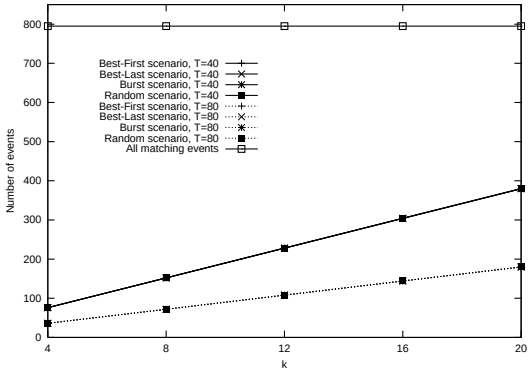
Subscribers generate subscriptions as follows. At random intervals, each subscriber generates m_S subscriptions. Each subscription is generated independently from the others and contains a random number of the following attributes: year, length, rating, votes, mpaa and genre. For each numerical attribute an operator must also be chosen. The operators we consider are $=, <$ and $>$. Operators are chosen according to predefined probabilities following a zipf distribution.

The value of each attribute can be generated using either a uniform or a zipf distribution. In the uniform method, string attributes take a value uniformly chosen over the set of all possible values. For example, for the mpaa attribute, all of the four possible values 'PG", "PG-13", "R" and "NC-17" have a 25% probability to be chosen. Numerical attributes take a value uniformly chosen between the minimum and maximum possible value of the attribute, according to the dataset.

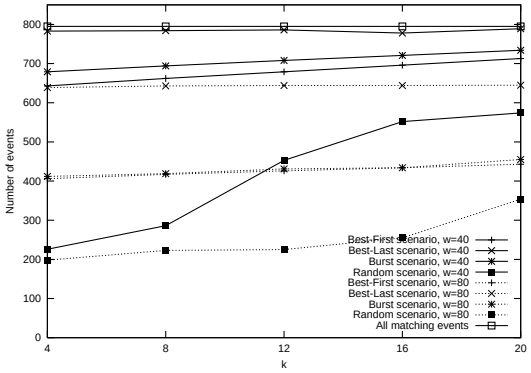
However, since real subscriptions are not expected to follow a uniform distribution, we also consider another method to generate subscriptions. This second method uses a zipf distribution to choose values for the various attributes. In this case, possible values for each attribute are ranked based on the dataset properties and our experience. For example, in a real scenario we would expect more people to subscribe for PG rated movies than for NC-17 rated movies, so the "PG" value is higher ranked than the "NC-17" value



(a) Continuous timing policy.



(b) Periodic timing policy.



(c) Sliding window timing policy.

Figure 6.2: Number of delivered events.

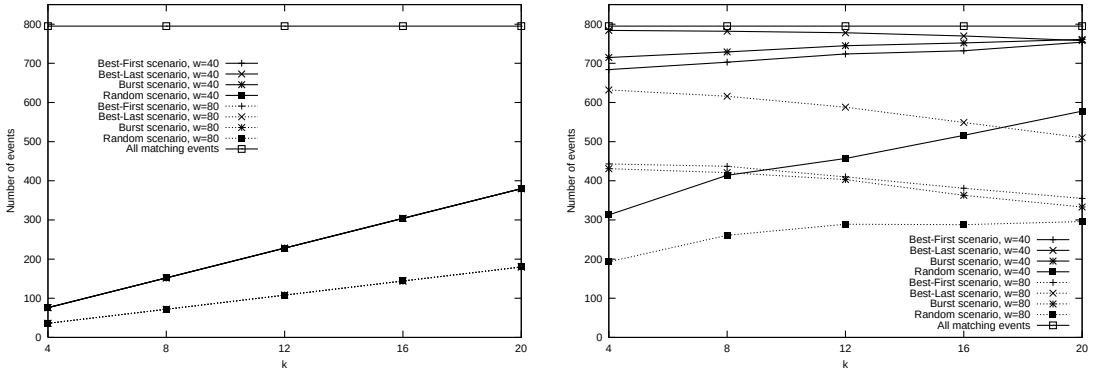
for the mpaa attribute. When a value for a given attribute must be chosen, the subscriber draws a number i from a zipf distribution and chooses the i^{th} most popular value for the attribute.

Finally, in any case, a preference rank uniformly distributed in $[0, 1]$ is associated with each subscription. Subscription examples can be seen in Figure 6.1b.

6.2 Experiments

We perform two sets of experiments. In the first set, we evaluate the number and quality of the events delivered to the users using PrefSIENA and SIENA. In the second set, we evaluate the overheads introduced by ranking.

Number and quality of delivered events: Since both the number and quality of events depend on the order of publications with regard to their ranks, to perform our experiments, we consider a number of different event-scenarios. In particular, in the "*Best-First*" scenario, the highest-ranked events are published first, while in the "*Best-Last*" scenario, these events are published after the lower-ranked ones. In the "*Burst*"



(a) Periodic timing policy.

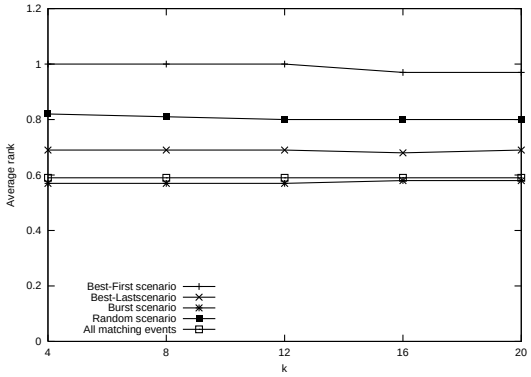
(b) Sliding window timing policy.

Figure 6.3: Number of delivered events, when diversifying.

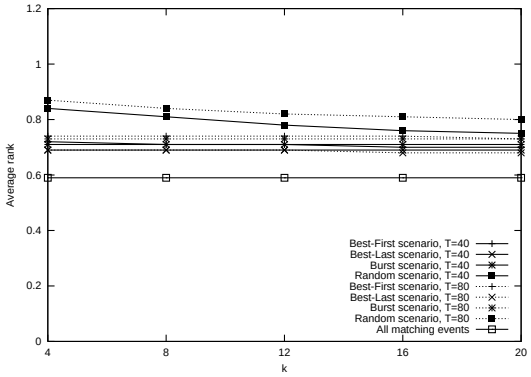
scenario, we consider the case of bursts of highly-ranked events at specific moments in time and finally, in the "*Random*" scenario, high and low ranked events are interleaved. For comparison, besides top-k delivery, we also consider the case in which all matching events are delivered to the users, as in traditional publish/subscribe.

First, we measure the number of events delivered to a specific subscriber using PrefSIENA as a function of the number k of the top results the subscriber is interested in. We use a set of 2500 events out of which 800 match the user's subscriptions. We first consider the continuous timing policy with no expiration (Figure 6.2a) and run this experiment for the above scenarios. The greatest reduction in received events occurs in the "*Best-First*" scenario. This happens because when low-ranked events arrive, they cannot enter the top-k results, since higher-ranked events already occupy all the available slots. The "*Best-Last*" scenario is the one with the most delivered events, since the user receives both the events with the lower ranks that arrive first and the events with higher ranks that arrive later. The number of delivered events increases along with k, something that is better illustrated in the "*Random*" scenario due to the mixed sequence of published events. In this case for example, for $k = 4$, PrefSIENA delivers on average 9% of the matching events, while for $k = 20$, the corresponding percentage is around 29%.

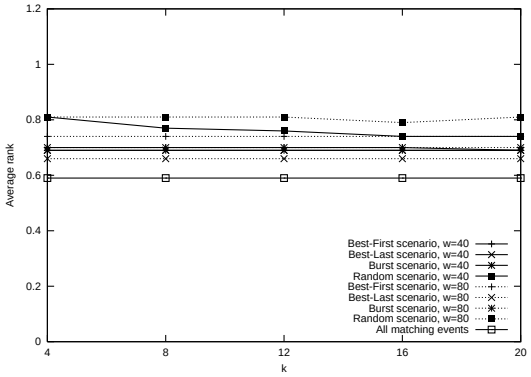
In Figure 6.2b, we show the number of delivered events for the periodic timing policy. We consider a constant rate of publications and run this experiment for periods with $T = 40$ and $T = 80$ events for all scenarios. The number of delivered events does not depend on the used scenario, since at each period this number is bounded by k. On average, the number of events delivered by PrefSIENA ranges from 7% to 35% of all matching events for the various values of k. The results for the sliding window timing policy are shown in Figure 6.2c. We use two window lengths, $w = 40$ and $w = 80$. In all scenarios, we observe that when $w = 80$, fewer events are delivered. This happens because when a larger window length is used, highly-ranked events remain in the subscriber's buffer for longer and prevent more low-ranked events from entering the top-k results. In the "*Random*" scenario, there are always some high-ranked events in the buffer that



(a) Continuous timing policy.



(b) Periodic timing policy.



(c) Sliding window timing policy.

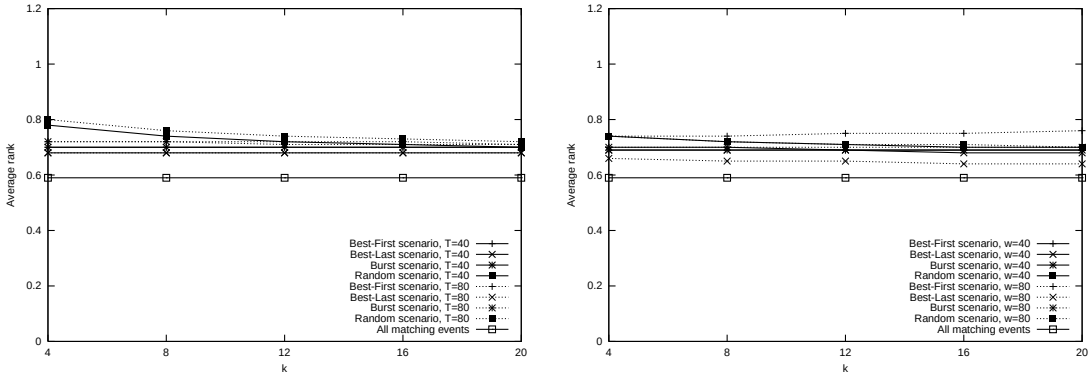
Figure 6.4: Average rank of delivered events.

block low-ranked ones from entering the top-k list. Therefore, the number of delivered events is considerably smaller than in the other scenarios. For example, for $k = 4$ the reduction is nearly 75%. Figure 6.3b shows the results for the sliding window timing policy, when diversifying the delivered events. Considering the various scenarios and the window length, the results are similar to the previous case. Generally, there is an increase in the number of delivered events, since more than one events may be now forwarded in each new window (see Section 4.2). This does not happen in the periodic policy, since the number of delivered events is the same, even if the actual events differ (Figure 6.3a).

In summary, periodic delivery ensures that each user receives a specific number of matching events per period. In contrast, in continuous and sliding window-based delivery, there is some fluctuation in the number of delivered events based on the order of event arrival with respect to their rank as well as on whether we require diversity or not. As expected, larger periods and windows decrease the event delivery rate.

We also run a set of experiments to evaluate the quality of the delivered events. We characterize quality based on three factors: (i) the average rank of delivered events, (ii) their diversity and (iii) their freshness, i.e. the elapsed time between their publication and the time they reach the user.

Figure 6.4 depicts the average rank of all the delivered events for the various timing



(a) Periodic timing policy. (b) Sliding window timing policy.

Figure 6.5: Average rank of delivered events, when diversifying.

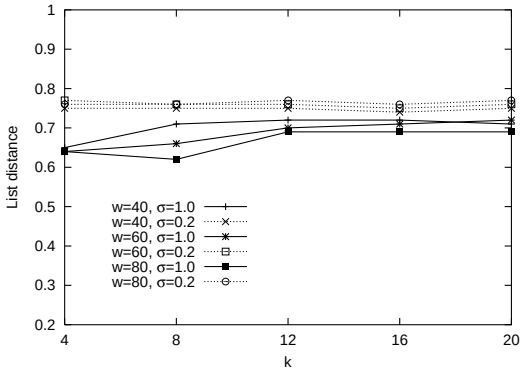
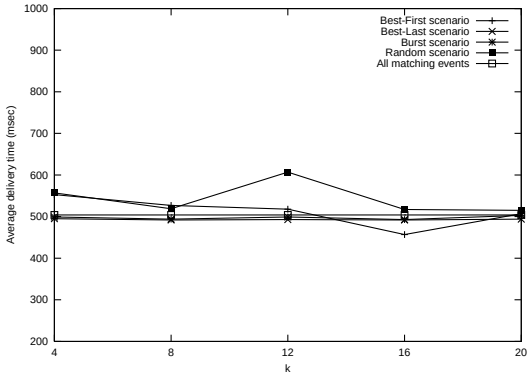


Figure 6.6: Sliding window timing policy: list diversity for delivered events.

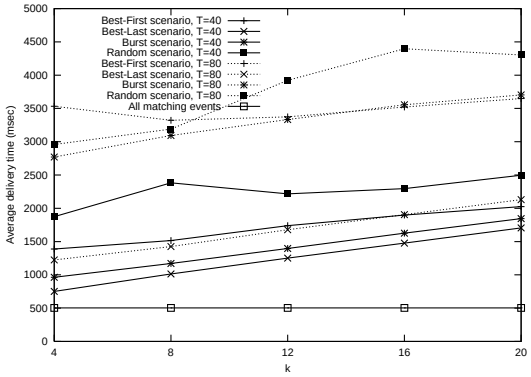
policies and scenarios. Generally, we observe that the average rank depends on the used scenario. The average rank of all matching events is 0.59. In PrefSIENA, even though in the presence of many high-ranked events some of them may fail to appear in the top-k results, the average rank is larger than that in all cases. When diversifying the events, there is a slight decrease of the average rank, since diverse events may have lower ranks (Figure 6.5).

In Figure 6.6, we measure the list-distance of the events that are forwarded to a user when we follow the sliding window timing policy for the "Random" scenario. We run this experiment for different window lengths using the diversification method with $\sigma = 1$ and $\sigma = 0.2$. We see that the produced results do indeed exhibit a higher diversity when they are chosen based not only on their ranks but also on their distance from each other. Similar behavior is observed in the periodic timing policy.

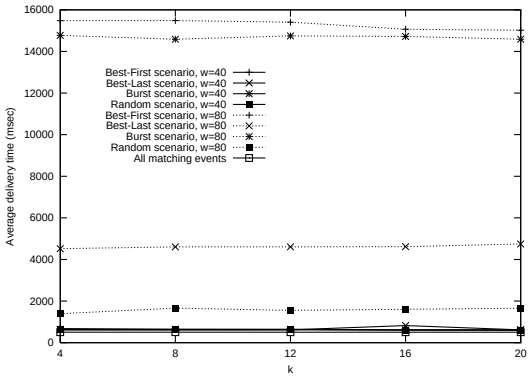
Next, we measure the freshness of the delivered events, that is the time between their publication and their delivery (Figure 6.7). In the continuous timing policy, the freshness of data does not depend on the scenario, since events are forwarded immediately. In the periodic policy, the sequence of the published events influences the freshness of the delivered ones. For example, if high-ranked events are published towards the end of a



(a) Continuous timing policy.



(b) Periodic timing policy.

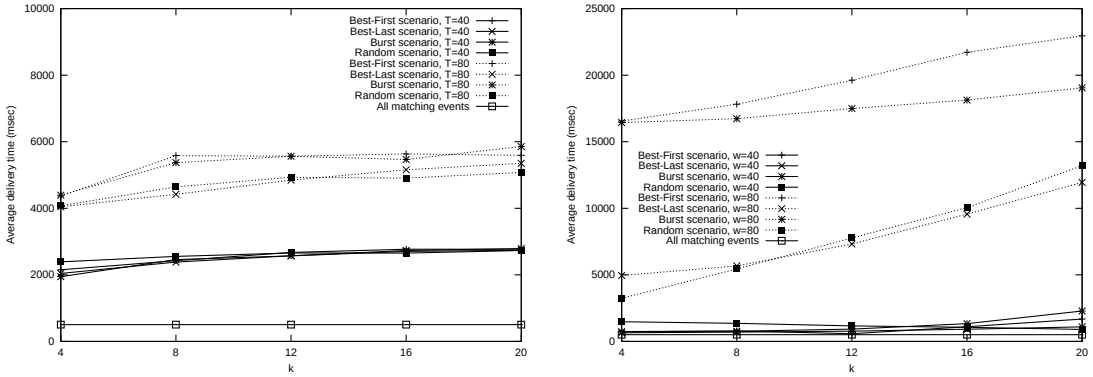


(c) Sliding window timing policy.

Figure 6.7: Freshness of delivered events.

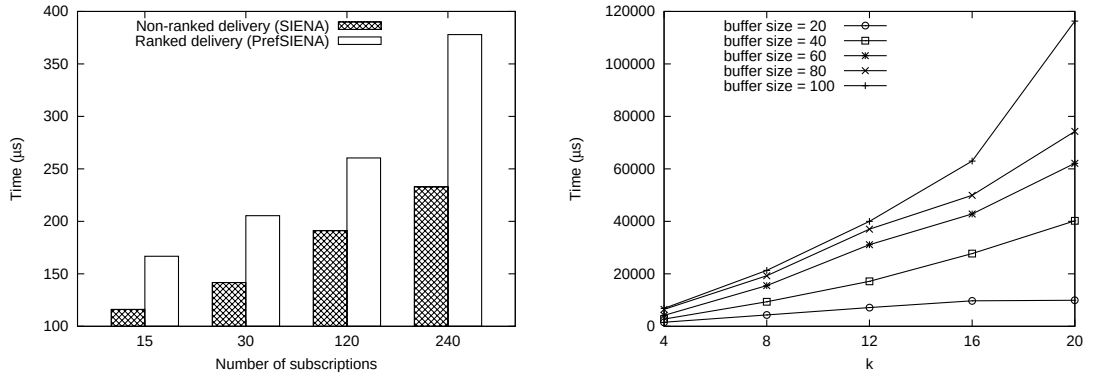
period, they will reach the user earlier than if they are published at the beginning. As expected, a larger period length results in larger delays between publication and delivery. In the sliding window timing policy, a larger window length increases the average delivery time. This happens because an event remains in the window for longer and therefore, it has more opportunities to enter the top-k. When we diversify the events, the average delivery time increases because of the additional time required to select which events to forward (Figure 6.8).

In summary, although top-k delivery reduces the number of delivered events, it increases their preference rank. All policies deliver events with comparable preference ranks. These ranks vary slightly with the order of event arrival; this variation is most noticeable in the continuous timing policy. Our diversification algorithm reduces the average preference rank of the delivered events, but increases their average diversity. Finally, in terms of freshness, the continuous policy is the most effective one. In the periodic and sliding window policies, freshness depends on the arrival rate and on the size of the period and the window respectively. Diversification reduces freshness mainly because of the additional overheads introduced by the diversification algorithm.



(a) Periodic timing policy. (b) Sliding window timing policy.

Figure 6.8: Freshness of delivered events, when diversifying.



(a) Matching time (per event). (b) Diversification time (per window).

Figure 6.9: PrefSIENA performance.

Performance: Finally, we perform a number of experiments to evaluate the performance of PrefSIENA. There is a substantial overhead for implementing ranked delivery of events for two reasons. First, to compute the importance of a new event, we have to locate all matching subscriptions, while in traditional publish/subscribe systems it suffices to locate just one of them. In Figure 6.9a, we evaluate this overhead. This depends on the size of the preferential subscription graph. On average, when performing ranked delivery, we have to check incoming events against twice as many nodes as in the case of non-ranked delivery. Second, there is also the overhead of maintaining state for previously forwarded events and performing computations to decide whether a new event belongs in the diverse top-ranked results. In Figure 6.9b, we measure the overhead of computing diverse top-k results using Algorithm 1 that implements our heuristic. The extra computational cost depends on the buffer size and the number k of produced results. We see that the required time increases linearly with both factors. Note that if instead of using the heuristic method for diversifying events we use the brute-force one, the required time increases much faster (for example, for a buffer size equal to 40 events and $k = 12$, this time exceeds 2 hours).

Chapter 7

Related Work

In this chapter we present related work on a number of fields related to our work. First, we describe some popular publish/subscribe systems that have been proposed in the literature and some forwarding techniques that have been applied. Then, we present recent work that focuses on ranking in publish/subscribe. Later, we describe various alternatives for users to specify preferences and also some work aiming to increase the diversity of results that are returned to users.

7.1 Publish/Subscribe

In this section we will describe a number of publish/subscribe systems. Most of the research so far in this field aims at improving the scalability of the system by decreasing the time required to perform matching between events and subscriptions. To do this, various indexing schemes have been proposed for storing subscriptions. These schemes depend on the way users define their subscriptions. There are two widely used methods to define subscriptions: the topic-based method and the content-based one. In the *topic-based* method (e.g. [22]), there are a number of predefined event topics, usually identified by keywords. Published events are associated with a number of topics. Users can subscribe to a number of individual topics and receive all events associated with at least one of them. Topic hierarchies can also be used with these method. When a user subscribes to some topic in the hierarchy the user implicitly also subscribes to all of its subtopics as well.

The topic-based method is easy to understand and implement, but has the disadvantage that the topics are static and predefined and therefore the users cannot express random interests. In the *content-based* method [13, 7], such as the one used in this work, the classification of the published events is based on their actual content. Users express their subscriptions through constraints which identify valid events. An event matches a subscription, if and only if, it satisfies all of the subscription's constraints. In general, the content-based method offers greater expressiveness to subscribers but is harder to implement. A third, not so widely used, alternative is the *type-based* method (e.g. [12]). This method is similar to the topic-based one with the difference that published events are not associated with keywords but rather with a type (which implies a certain event structure).

Publish/Subscribe Systems: A centralized, main memory mechanism for matching incoming events against a set of stored subscriptions is described in [13]. The proposed method is processor cache conscious and makes use of the "prefetch" command which is available in modern processors to fetch certain blocks from memory and thus achieve better performance. Subscriptions are treated as sets of predicates, as in our work. Each predicate consists of an attribute a, a value v and a relational operator op . An event is a set of pairs, each one of them consisting of an attribute a' and a value v'. An event pair (a', v') matches a subscription predicate (a, v, op) if $a' = a$ and v' op v.

The main idea behind the proposed matching algorithm is to cluster subscriptions according to their predicates and define an *access predicate* for each cluster. An event can match some of the subscriptions of a cluster only if it matches the cluster's access predicate. Since an incoming event is expected to match only a small number of the defined access predicates, we only have to check a small number of clusters to find the subscriptions that are matched by the incoming event. More specifically, the proposed algorithm makes use of a list of clusters (each one associated with an access predicate), an index on those access predicates and a bit vector. The bit vector consists of one bit for every predicate known to the system. Upon receiving a new event, the bit vector is reset. For each predicate contained in the event, the corresponding bit is set to 1. Using the index, the access predicates that match the incoming event are found. After this, the subscriptions of the corresponding clusters are checked one by one against the bit vector to determine whether they contain any predicates that are not satisfied by the event. Only equality predicates common to all the subscriptions of a cluster are used as access predicates. The most challenging part of the method, however, is to define those access predicates in such a way as to minimize the number of clusters that each incoming event has to be matched against. The authors propose a cost model based on their implementation choices to calculate the cost of matching an incoming event against a certain configuration of the system (number of clusters etc.) and use a greedy algorithm to find a local cost optimum.

Experimental evaluation shows that the proposed algorithm is highly scalable, support-

ing millions of subscriptions and high rates of incoming events. The proposed algorithm is very fast indeed but remains centralized and therefore prone to failure. However, it can be applied to each of the nodes of a distributive system as in the case of COBRA [24]. Also, since the proposed algorithm is a main memory one, for it to be fast, the node running it should have enough memory available to store all of the subscriptions and events it is aware of. Another issue is the use of processor commands which means that the algorithm implementation is processor-specific.

Scribe [9] is a topic-based publish/subscribe system. It is built on top of Pastry [25], a peer-to-peer routing protocol. Each Pastry node has a unique id. These ids are uniformly distributed. Pastry offers one main operation: Given a message and a key, it routes the message to a node such that the node's id is numerically closest to the key among all the other ids. Ids and keys are both sequences of digits with base 2^b (where b is a system parameter). The routing of a message needs $\lceil log_{2^b} N \rceil$ steps on average, where N is the number of nodes in the system. Delivery is guaranteed even in the case of many simultaneous node failures. The routing table of each Pastry node X has $\lceil log_{2^b} N \rceil$ rows, each of which contains $2^b - 1$ entries. All entries in row n of the table refer to nodes whose ids match X's id in the first n digits and whose $(n+1)^{th}$ digit differs. Generally, to route a message along with its key, the current node forwards the message to a node whose id shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the current node's id.

Scribe makes use of Pastry's routing strategy to built an application-level multicast infrastructure on top of it. A scribe node can create groups. Other nodes can later join these groups or multicast messages to them. Scribe is responsible to deliver the multicasted messages to the appropriate nodes. There exists one group (with a unique group id) for every available topic. The Scribe node with id numerically closest to the group id is responsible for the group. This node is called the *rendezvous point* for the group. The rendezvous point is the root of a multicast tree which includes all the members of the group. Such trees are created by joining the Pastry routes from each group member to the rendezvous point and may therefore contain nodes that are not members of the group. When a group member wants to publish a new event associated with a specific topic, it forwards the event to the corresponding rendezvous point responsible for the topic. From there, the event is delivered to all interested parties via the corresponding multicast tree. Scribe takes advantage of the scalibility and fault-tolerance of the Pastry protocol but routes all group messages via a single rendezvous point. Also, it is a DHT-based approach, and therefore inherits all of the DHT's maintenance costs.

In [24], the problem addressed is the creation of a customized RSS (Really Simple Syndication) feed for a user via the aggregation of the vast number of RSS feeds that are available on the Internet. Towards this direction, the Cobra (Content-Based RSS Aggregator) system is presented. Motivated by the rapid growth of continuously-updated discussions in the blogoshere, the authors propose a three-tiered network of *crawlers*,

filters and *reflectors* to tackle the problem of finding interesting blog posts and track blogs with interesting content. The purpose of the Cobra system is to gather posts (crawlers), perform content-based filtering for each of its subscribers (filters) and present to them a personalized RSS feed (reflectors). Cobra includes an offline service provisioning technique that determines the components needed to support a certain number of sources and subscribers, i.e. it determines the number of crawlers, filters and reflectors used by the system. These components can be distributed over a number of hosts.

Each crawler is given a (separate) list of source blog URLs and periodically crawls each URL to find new posts. Each URL is assigned to a crawler according to latency measures. In order to reduce bandwidth consumption, crawlers make use of http meta data and a hashing technique to distinguish between old and new posts. New posts are pushed to the second tier, the filters. Each filter receives posts from all the crawlers. A filter contains a list of subscriptions of the form (*subscription id*, *reflector id*, *keywords*). The subscription id is unique for every subscription in the network. The reflector id identifies the reflector responsible for the subscriber who issued the subscription. In order for a subscription to match a published post, all of its keywords must be found inside the post. In order to perform the matching, each Cobra filter uses the algorithm proposed in [13] because of its high scalability and efficiency. The new posts which match a subscription are forwarded to the corresponding reflector. A reflector is responsible for collecting all the relative posts for a number of subscribers and create a personalized RSS feed for each one of them. Due to performance reasons, filters only push the matched posts to the reflectors (without a subscriber list attached). Therefore, each reflector re-runs the filtering algorithm for each post it receives to decide the subscribers who should receive it. For each user subscription, the corresponding reflector caches the k latest posts received and maintains a feed which the user can view via any RSS reader.

Periodically, Cobra makes use of statistics collected at each component to re-run the offline provisioning algorithm at a central controller node and adjust the number of needed components. Cobra can be used along with one of the many already existing RSS readers and uses the already very popular RSS feeds as sources. However, it is not a fully distributed system as it is designed to run on a hosting center.

Forwarding Techniques: The work in [22] is focused on topic-based publish/subscribe systems. Concerned by the cost introduced by these systems in order to maintain a supporting structure (such as a multicast tree) for each distinct topic, the authors propose a distributed clustering algorithm that makes use of the correlations between the subscriptions stored in the system to group topics together into *virtual topics*. The corresponding supporting structures are then unified, thus reducing the overall cost of the system.

The authors assume a basic topic-based system where users subscribe to a number of topics. For each topic there is a node in the system which is considered to be responsible for it. For each topic there exists a multicast tree rooted at the corresponding responsible node. This tree connects all the users who have subscribed to the corresponding topic.

Whenever a publisher publishes an event on this specific topic, it sends it to the responsible node. Using the multicast tree the event eventually reaches all the subscribed users. The main idea of the proposed method is to group topics with similar sets of subscribers into a virtual topic and merge their supporting structures, namely their multicast trees. Users can use a local filter to prune irrelevant results in case they are interested in only a fraction of the virtual topic's topics.

A cost model for the structure maintenance of the system and the dissemination of events is introduced. The maintenance cost depends on two factors: the average maintenance cost for a multicast tree with a given number of subscribers and the cost of maintaining the association between a given virtual topic and each of its topics. The dissemination cost also depends on two factors: the cost of notifying the responsible node of a topic (or a virtual topic) about an event and the cost of propagating the event through the multicast tree to the appropriate subscribers. The overall cost is the sum of all these factors over all the existent topics and virtual topics.

To form and maintain virtual topics only local operations are performed. Those operations involve only the nodes participating in one or two topics (or virtual topics) and include the grouping of two topics to form a virtual topic, the addition of a topic to an existing virtual topic, the merging of two virtual topics, the removal of a topic from a virtual topic and the destruction of a virtual topic. Prior to the performance of such an operation, its contribution to the reduction of the overall cost is estimated. The operation is performed only if it is beneficial to the system. Since it is not practical to estimate the overall cost based on all the existing topics, only the participating topics are used (for example, when merging two virtual topics we estimate the overall cost reduction based on these two virtual topics alone). For a more detailed explanation of each operation the reader is referred to [22]. Finally, the authors propose a heuristic greedy algorithm for a subscriber to find a combination of topics and virtual topics that cover its interests. The proposed methods have been implemented on top of Scribe [9].

A forwarding algorithm for content-based publish/subscribe systems is presented in [17]. Since content-based matching does not have a worst case efficient solution [18], the authors attempt to reduce the number of content-based matching operations needed. The basic idea of their method is to perform only one content-based matching per event when the event enters the system and associate the event with a *prefix*. This prefix is then used by all other nodes to route the event.

Routing is performed in a SIENA-like fashion [7]. SIENA's poset data structure is replaced by the Routing Tree. All servers of the system maintain a Routing Tree. The Routing Tree stores filters in a disjoint form. Each of its nodes hold a simple attribute constraint with only one predicate. Some nodes of this tree contain pointers to subscribers while only new root nodes of the tree are propagated to other servers of the network. When a new event reaches a server it is matched against its Routing Tree. The result of this operation is a Forwarding Prefix Tree, which is constructed by the nodes and paths of the Routing Tree that match some of the event's attributes. This Forwarding Prefix Tree

is attached to the event and piggy-backed along with it toward the appropriate subset of the server's neighbors. When a server receives an event along with a Forwarding Prefix Tree, it maps the Forwarding Prefix Tree on its own Routing Tree to find the appropriate neighbors that it must forward the event to.

A major drawback of the proposed method is that in order for it to work, all servers must have identical Routing Trees (possibly with different neighbors stored at their nodes). For this to happen all Routing Trees must be composed of the same set of filters and those filters must be inserted into the trees at the same order. This is nearly impossible to happen in a distributed environment where each subscriber sends its subscriptions (filters) to only a very small number of servers. To overcome this obstacle, the authors propose the use of the Tree Optimizer. The Tree Optimizer is one central node in the system which is responsible for the construction of the Routing Tree and its propagation to all the other nodes. A possible extension is the use of many nodes, each of which will be responsible for a part of the Routing Tree. Servers must forward all newly received filters that do not already exist in their Routing Tree to the Tree Optimizer. Since it would be impractical to wait for the changes to the Routing Tree to be propagated to the whole network, they insert pointers to the subscribers in their own (now outdated) copy of the Routing Tree but only in already-existing nodes. This results in a number of false positives until the changes in the Routing Tree are propagated to the network and the local Routing Table copy becomes more precise.

The proposed method lowers the number of content-based matching operations to only one per event but inserts the overhead of the Forwarding Prefix Tree to each event and the notion of the Tree Optimizer which remains mainly a centralized structure taht inserts significant maintenance costs.

Another content-based routing technique, described in [8], aiming to increase scalability. The authors make use of a fundamental concept of content-based routing, i.e. events and subscriptions must "meet" at some point in the network. They propose to route events and subscriptions on different but intersecting partitions of the network. The routing infrastructure is perceived as a multidimensional grid. Subscriptions are disseminated through partitions that cover the whole network but do not overlap with each other. Events are disseminated through partitions that intersect all subscriptions partitions. To visualize this concept, assume a 2-dimensional grid. A possible partitioning of this space is to disseminate subscriptions along the columns of the grid and events along its rows. Subscriptions are routed in a SIENA-like fashion but can be propagated only on a single partition of the network. For example, if a new subscription s is issued by X in the 2-dimensional space, it will be propagated only on X's (entire) column. If s is also issued by some other node Y of the same column, then it will be propagated along the column until it reaches a node that is already aware of it because of X's previous subscription. From that point on, s does not need to be propagated again and therefore no additional resources are needed for the handling of the second subscription. Simulation studies show that the proposed technique can indeed increase scalability.

7.2 Ranked Publish/Subscribe

In this section we review related work on ranked publish/subscribe system. Up to now, there has not been much research in this field. To the best of our knowledge, the following work, along with our own, is the first on this topic.

In [21], the problem of ranked publish/subscribe systems is also considered. However, the problem is viewed in a different way. In a sense, the authors consider the "reverse" or "dual" problem, since they aim to recover the most relevant matching subscriptions for a published event (instead of locating the most relevant events for each subscription). Such a view of the problem is interesting in applications such as targeted web advertising, where as an "event" we consider a user's visit to a website and the advertisers wish to display their advertisements ("subscriptions") only to the users that are the most interested in them. Subscriptions are modeled as sets of interval ranges in a number of dimensions and events as points that match all the intervals that they stab. Each interval is associated with a predefined score. Top subscriptions are recovered based on those scores. A subscription matches an event exactly, if and only if, the event is fully contained the subscription's hyper-rectangle. If the event stabs only some of the subscription's intervals then we have relaxed matching. In the first case, each subscription is associated with a single score while in the latter case each interval of the subscription is associated with a weight and the score of the subscription is computed based on the weights of the stabbed intervals.

To quickly recover the top subscriptions related to an event, a scored interval index is build over the subscription intervals of each possible dimension. Those indices take in an event value v_i and provide an iterator returning the intervals containing v_i in the order of their score. Given an incoming event $(v_1, \ldots, v_n)$, the n corresponding indices are probed. Then, in the case of exact matching, the produced intervals are intersected to produce the subscriptions in score order. In the case of relaxed matching, the Threshold Algorithm [14] is used to find the highest score subscriptions. After a study of existing tree structures like interval trees, segment trees and R-trees, the authors also propose two new tree structures to index each of the mentioned dimensions: the interval R-tree (IR-tree) and the Score-Optimal R-Tree ($SOPT$-R-tree). The IR-tree uses an R-tree instead of a list to index the intervals of each node of an interval tree, while the $SOPT$-R-tree is in fact a scored R-tree in which the intervals are sorted in a careful way so that less tree nodes are accessed by the tree's iterator.

A problem that arises in this approach is that a user (who is treated as an event) that does not match subscriptions with a high score receives data depending on the existence of other users. For example, a user X that does not belong to the advertiser's target group will not receive any advertisements as long as there are other users who do. However, if there are no such users, X will receive those same advertisements.

Another work that also deals with the problem of ranked publish/subscribe is [23]. In the proposed model, a subscriber receives the k most relevant events per subscription

within a window w which can be either time-based or number-based. Both events and subscriptions are associated with expiration times. Event relevance to a subscription is measured as the event's distance to the subscription, as computed by a user defined ranking function. All events that are among the user's top-k results at some point in time will eventually be delivered to the user.

Events are divided into three groups: (i) Excellent candidates, i.e. events among the user's top-k results at the moment of their publication, (ii) Good candidates, i.e. events that are not among the user's top-k results at the moment of their publication but have a probability larger than some threshold σ to enter the top-k results before the current window passes and (iii) Bad candidates, i.e. events for which this probability is less that σ. For each user subscription a queue is maintained. This queue has a head for keeping excellent candidates and a tail for good candidates. The focus is on efficiently maintaining this queue. The size of the head is k while the size of the tail depends on σ. The authors assume Poisson distributions for the generation and expiration of events and compute a minimum length for the queue's tail so that all good candidates that have high probability to enter the user's top-k results at some point in the future can be stored there. This length grows sub-linearly with k.

7.3 Preferences

In this section, we describe the way users can express various degrees of interest for their subscriptions by using preferences. There are two different approaches for expressing preferences, the quantitative and the qualitative one. In the *quantitative approach* (e.g. [6, 20]), preferences are expressed indirectly via the employment of scoring functions. Such functions associate a numeric score with each specific data item. Those scores indicate the user's interest for the corresponding data. In [6] for example, users can assign to each item a score in $[0, 1]$ to indicate interest, declare their indifference in it or even veto it from ever appearing in their results.

In the *qualitative approach* (e.g. [10, 19, 15]), preferences between two data items are specified directly by the users, typically using some form of binary relations between data. In [15] for example, qualitative preferences are used to express priorities among the values of specific attributes of relational databases and also among the attributes themselves. Based on those preferences, a query lattice is constructed and utilized to retrieve database tuples in the order directed by the preferences defined by the user.

The qualitative approach is more general than the quantitative one, since scoring functions can always be defined in terms of binary relations. However, not all binary relations among data items can be captured by scoring functions. Therefore, the qualitative approach offers greater expressiveness to users.

User preferences can also be combined so that additional preferences can be extracted. Since different preferences may follow different orders (such as the strict partial, weak or total order), there are various ways to combine them that can either preserve those

orders or not [10]. Such ways include the prioritized, pareto and lexicographic preference compositions.

Since user preferences may depend on the user's current state, *contextual preferences* have been introduced to further increase user expressiveness. Contextual preference models have been proposed following both the quantitative ([27, 26]) and the qualitative ([5, 16]) approach. When contextual preferences are defined by the users, an extra preprocessing step is required to select the appropriate subset of preferences that apply under the user's current context. Further computations should be based only on the selected preferences.

7.4 Diversity

The notion of diversity is lately beginning to be considered in personalized systems. Up to now, most of research focused on improving the recommendations made to the users by the various personalized systems. However, while accurate suggestions are fundamental for those systems, they do not always guarantee user satisfaction. Diversity aims to produce recommendation lists that, when viewed as a whole, are more satisfactory to the users.

In [29], the authors propose a method for topic diversification, i.e. a method for modifying personalized recommendation lists in order to increase user satisfaction. The purpose is to balance recommendation lists so that they do not include only suggestions related to the user's top-ranked interests but reflect their complete spectrum of interests instead, while still continuing to take into consideration the accuracy of the individual suggestions. The intra-list similarity metric is introduced to assess the topical diversity of a given recommendation list. The proposed method can be applied on top of any recommendation list produced with any method (usually some form of collaborative filtering), as long as its size is larger than the desired size of the final list.

Assuming a set B of items and an arbitrary function $c : B \times B \rightarrow [-1, +1]$ measuring the similarity $c(b_i, b_j)$ between items b_i, b_j, the *intra-list similarity* of a recommendation list P is defined as:

$$ILS(P) = \frac{\sum_{b_i \in P} \sum_{b_j \in P, b_i \neq b_j} c(b_i, b_j)}{2}$$

Higher intra-list similarity scores denote lower diversity of the corresponding list. Note that the metric is permutation-insensitive. In order to produce the final diversified list P', we first insert into it the first item of P. Then, for every subsequent item we want to insert in the z^{th} position of P', we collect the items of P that do not occur in positions 0 to z in P' and compute their similarity with those that do. Sorting these items in reverse order according to the similarity we computed, we obtain a dissimilarity rank for each of them. This dissimilarity rank is merged with the original recommendation rank that the object has in P, yielding a final rank for the item. The highest ranked item is then inserted into P'. While the diversification of a recommendation list naturally leads to the

decrease of the precision and recall of its suggestions, a wide case study with more than 2000 users showed that user satisfactions increases.

The notion of diversity is also explored in [28], where the authors focus on queries concerning e-shopping. Queried items are tuples of a database relation R. Motivated by the fact that some relation attributes are more important to the user, a method is proposed where a recommendation list is diversified by first varying the values of higher priority attributes before varying the values of lower priority ones. Therefore, having defined a diversity ordering, i.e. an ordering $\prec_R$ of R's attributes, the authors then define a prefix with respect to $\prec_R$ as a sequence of attribute values, in the order given by $\prec_R$, moving from highest to lower priority. If two tuples t_i, t_j share a prefix ρ of length l, then their similarity $SIM(t_i, t_j)$ is 1 if their $(l+1)^{st}$ attribute is the same and 0 otherwise. A set S of tuples is defined to be diverse with respect to ρ if $SIM(t_i, t_j)$ is minimized over all pairs of tuples in S. In case the tuples are associated with scores, the scored variation of diversity always picks tuples with higher scores over tuples with lower ones. If many tuples are tied, then the tuples are picked in a diversity preserving way.

Chapter 8

Conclusions And Future Work

In this thesis, we extend the publish/subscribe paradigm with a ranking mechanism so that only the top-ranked events are delivered to each user. Ranking is based on letting users define their preferences among specific events. Our overall goal in this work has been to increase the quality of events received by the users of publish/subscribe systems in terms of: (i) their importance or relevance, (ii) diversity and (iii) freshness. Ranking events by importance is achieved by letting users express preferences along with their subscriptions. Events that match more preferable subscriptions are ranked higher than events that match less preferable ones. To rank an event, we also take into account how different the event is from the other top-ranked ones so that the overall diversity among the delivered event notifications is increased. Finally, for freshness, we have examined a number of policies with regards to the time range over which the top-k events are computed, namely a continuous, a periodic and a sliding window one. We organize preferential subscriptions in a graph and utilize it to forward events to users. We have fully implemented our approach in SIENA, a popular publish/subscribe middleware system.

Our overall focus has been on increasing the value of the events received by each user. There are many directions for future work, mainly regarding performance. One is developing indexing structures towards making matching events with subscriptions and ranking more efficient. Also, organizing the servers of the event-notification service in other topologies besides the hierarchical one used in this work and exploit the underlying structures to further reduce the messages propagated through the network. Regarding expressiveness, an interesting direction is the extraction of preference ranks for published events based on more than one factors in a skyline fashion. In this case, diversity can also be employed to resolve ties among the events that belong to the skyline.

Bibliography

[1] *The Internet Movie Database*. http://www.imdb.com.

[2] *Movies dataset*. http://had.co.nz/data/movies.

[3] *PrefSIENA*. http://www.cs.uoi.gr/∼mdrosou/PrefSIENA.

[4] *SIENA*. http://serl.cs.colorado.edu/∼serl/dot/siena.html.

[5] R. Agrawal, R. Rantzau, and E. Terzi. Context-sensitive ranking. In *SIGMOD Conference*, pages 383–394, 2006.

[6] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD Conference*, pages 297–306, 2000.

[7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[8] S. Castelli, P. Costa, and G. P. Picco. Hypercbr: Large-scale content-based routing in a multidimensional space. In *INFOCOM*, 2008.

[9] M. Castro, P. Druschel, A. marie Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20, 2002.

[10] J. Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[11] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[12] P. T. Eugster and R. Guerraoui. Content-based publish/subscribe with structural reflection. In *COOTS'01: Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems*, pages 10–10. USENIX Association, 2001.

[13] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD Conference*, pages 115–126, 2001.

[14] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[15] P. Georgiadis, I. Kapantaidakis, V. Christophides, E. M. Nguer, and N. Spyratos. Efficient rewriting algorithms for preference queries. In *ICDE*, pages 1101–1110, 2008.

[16] S. Holland and W. Kie ling. Situated preferences and preference repositories for personalized database applications. In *ER*, pages 511–523, 2004.

[17] Z. Jerzak and C. Fetzer. Prefix forwarding for publish/subscribe. In *DEBS*, pages 238–249, 2007.

[18] S. Kale, E. Hazan, F. Cao, and J. P. Singh. Analysis and algorithms for content-based event matching. In *ICDCS Workshops*, pages 363–369, 2005.

[19] W. Kiessling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

[20] G. Koutrika and Y. E. Ioannidis. Personalized queries under a generalized preference model. In *ICDE*, pages 841–852, 2005.

[21] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. In *VLDB*, 2008.

[22] T. Milo, T. Zur, and E. Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In *SIGMOD Conference*, pages 749–760, 2007.

[23] K. Pripuzic, I. P. Zarko, and K. Aberer. Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w. In *DEBS*, pages 127–138, 2008.

[24] I. Rose, R. Murty, P. R. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh. Cobra: Content-based filtering and aggregation of blogs and rss feeds. In *NSDI*, 2007.

[25] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[26] K. Stefanidis and E. Pitoura. Fast contextual preference scoring of database tuples. In *EDBT*, pages 344–355, 2008.

[27] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Adding context to preferences. In *ICDE*, pages 846–855, 2007.

[28] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.

[29] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW*, pages 22–32, 2005.

Author's Publications

Marina Drosou, Evaggelia Pitoura and Kostas Stefanidis, *Preferential Publish/Subscribe*, in Proc. of the 2nd International Workshop on Personalized Access, Profile Management and Context Awareness: Databases (PersDB 2008), in conjunction with the VLDB 2008 Conference, August 23, 2008, Auckland, New Zealand.

Marina Drosou, *XML Summaries for Routing in P2P systems*, in Proc. of the 1st Panhellenic Scientific Student Conference in Informatics, Computer Engineering and Related Technologies (EUREKA 2007), May 18-20, 2007, Patras, Greece.

Short Vita

Marina was born in Ioannina in 1984. She was admitted at the Computer Science Department of the University in Ioannina in 2002 and graduated in 2006. At the same year she began her postgraduate studies at the same department. She is a member of the Distributed Management of Data (DMOD) Laboratory since 2006. Her research interests include publish/subscribe systems and sensor networks.