ProMo - A Scalable and Efficient Framework for Online Data Delivery

Haggai Roitman, Avigdor Gal
Technion - Israel Institute of Technology
Haifa
32000 Israel
{haggair@tx, avigal@ie}.technion.ac.il

Louiqa Raschid
University of Maryland
College Park
MD 20742 U.S.A
louiqa@umiacs.umd.edu

1. INTRODUCTION

Web enabled application servers have had to increase the sophistication of their server capabilities in order to keep up with the increasing demand for client customization. Typical applications include RSS feeds, stock prices and auctions on the commercial Internet, and increasingly, the availability of Grid computational resources. Web data delivery technology has not kept up with these demands. There still remains a fundamental trade-off between the scalability of both performance and ease of implementation on the server side, with respect to the multitude and diversity of clients, and the required customization to deliver the right service/data to the client at the desired time.

Current data delivery solutions can be classified as either push or pull solutions, each suffering from different drawbacks. Push (*e.g.*, [6]) is not scalable, and reaching a large numbers of potentially transient clients is typically expensive in terms of resource consumption and implementation by a server. In some cases, where there is a mismatch with client needs, pushing information may overwhelm the client with unsolicited information. Pull (*e.g.*, [5, 1]), on the other hand, can increase network and server workload and often cannot meet client needs. Several hybrid push-pull solutions have also been presented in the past [2].

$\mathcal{P}ro\mathcal{M}o$ is a framework that includes a language, model, and algorithms to support flexible, efficient and scalable targeted data delivery. It is flexible since it can accommodate push, pull or hybrid push-pull solutions. It is efficient since it exploits server capabilities in meeting client demands.

$\mathcal{P}ro\mathcal{M}o$ provides a uniform specification language that can be used by clients to specify client data needs and that can be used by servers to specify server data delivery capabilities. In $\mathcal{P}ro\mathcal{M}o$, each client (or server) specification is specified *independent* of other specifications, and is *transparent* to the current data delivery solution. Within this framework, which is illustrated in Figure 1, a proxy for the client takes the responsibility to match client needs with relevant server capabilities. The proxy determines if the server's capabilities can satisfy (or partially satisfy) the client's needs. Further, the proxy generates a data delivery schedule that exploits the available server capabilities. To do so, the proxy will exploit server push, and as needed, it may augment server push with pull actions.

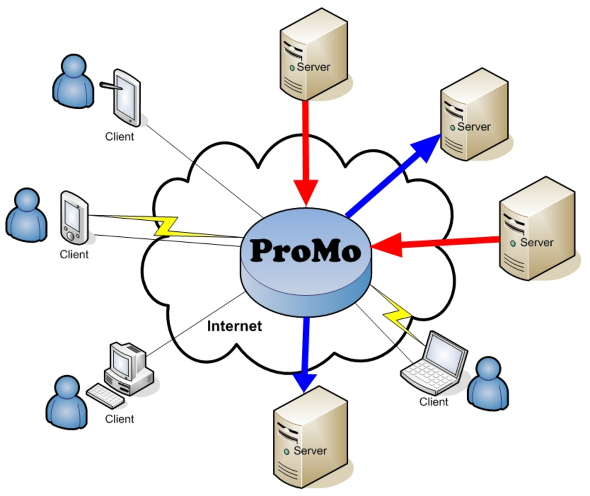
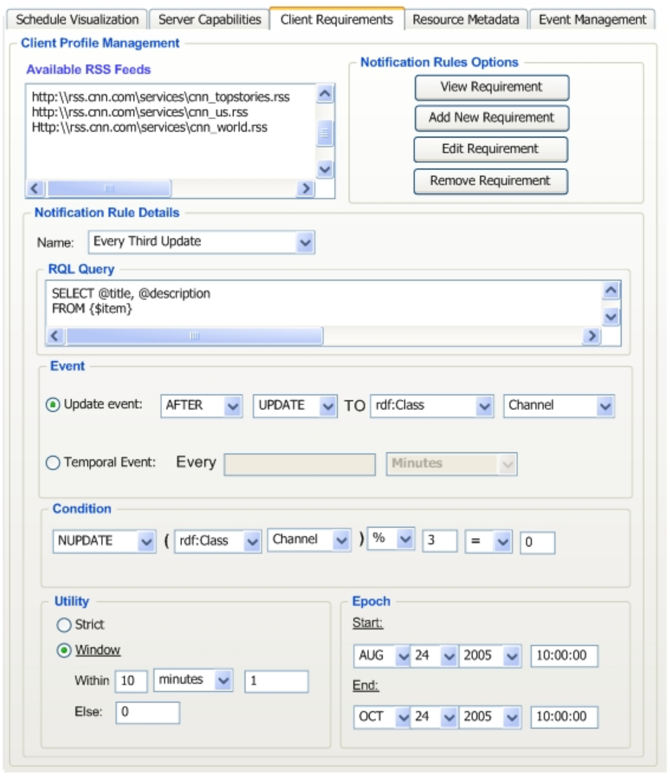


Figure 1: Promo Framework

In this demonstration presentation, we describe the $\mathcal{P}ro\mathcal{M}o$ framework. We first discuss the interface (GUI) to specify a client profile. The same interface and language can be used to specify server capabilities. We use the example of a profile specified for an RSS feed [10]. We then describe the components of the $\mathcal{P}ro\mathcal{M}o$ framework. Finally, we show the visualization GUI of the $\mathcal{P}ro\mathcal{M}o$ framework. For a chosen profile and updates at the server, we illustrate the $\mathcal{P}ro\mathcal{M}o$ schedule and show how $\mathcal{P}ro\mathcal{M}o$ will exploit push from the server and augment with pull actions. The implementation is in Java JDK 1.4.2.

2. PROFILES AND CAPABILITIES

In the $\mathcal{P}ro\mathcal{M}o$ framework, a client can specify a profile for data delivery needs using an expressive specification language that includes notification rules. Notification rules are also utilized to specify server capabilities. The user interface for a client profile specification is given in Figure 2. Figure



Available RSS Feeds
http:\\rss.cnn.com\services\cnn_topstories.rss
http:\\rss.cnn.com\services\cnn_us.rss
Http:\\rss.cnn.com\services\cnn_world.rss

Notification Rule Details
Name: Every Third Update

RQL Query
SELECT @title, @description
FROM {$item}

Event
[✓ Update event:] AFTER UPDATE TO rdf:Class Channel
Temporal Event: Every Minutes

Condition
NUPDATE (rdf:Class Channel) % 3 = 0

Utility
Strict
[✓ Window]
Within 10 minutes 1
Else: 0

Epoch
Start: AUG 24 2005 10:00:00
End: OCT 24 2005 10:00:00

Figure 2: Profile Specification Interface

2 illustrates the settings corresponding to the following profile expressed in English: *"Return the title and description of items published on the CNN Top Stories RSS feed channel, once three new updates to the channel have occurred. Notifications received within ten minutes after new three update events have occurred will have a utility value of* 1*. Notification outside this window have a value of* 0*. Notifications should take place during two months starting on August 24th 2005 , 10:00:00 GMT."*

A Profile p contains two elements, namely *Domain* and *Notification*. $Domain(p)$ is a set of resources $\{R_1, R_2, ..., R_n\}$ of interest to the client. A notification is a rule defined over a subset of resources in $Domain(p)$. A Profile contains one or more notification rules.

Using the interface of Figure 2, the user can choose a set of available resources; resources are defined as classes and attributes in the RSS specification. Next, the user will create a set of notification rules over these resources. For each notification rule, the user can specify the following:

- An RQL [7] query Q that should be executed;
- A trigger expression which specifies the triggering event and the condition that must be satisfied so that the rule should be evaluated;
- An epoch during which notifications of Q should be delivered to the client;
- The utility the client gains from such notifications.

For example, the profile given in Figure 2 references three RSS feeds and specifies one notification rule. For the server capability, the user interface allows the server to identify the type of supported events (*e.g.*, update-based events or temporal events), conditions for notifications and an update policy. The update policy is often referred to as *life*. The *life* policy can have a value `Overwrite` where an update history is kept until a new event occurs and overwrites the prior event. Alternately, a *life* policy can have a `Window Y`, where `Y` is the width of the window. The $\mathcal{P}ro\mathcal{M}o$ profile language can be found at [8].

3. $\mathcal{PROMO}$ FRAMEWORK OVERVIEW

Figure 3 describes the four main components of $\mathcal{P}ro\mathcal{M}o$; they are the network layer, profile management, model management and schedule management.

Network Layer All interactions between the $\mathcal{P}ro\mathcal{M}o$ proxy and clients or servers are done via TCP/IP connections. Both clients and servers submit their profiles to the $\mathcal{P}ro\mathcal{M}o$ proxy; a $\mathcal{P}ro\mathcal{M}o$ profile is expressed using XML. This layer is further used for client notifications (marked as yellow lightning in figures 1 and 3) and during schedule execution. Schedule execution can be classified into two cases; first, when the $\mathcal{P}ro\mathcal{M}o$ proxy monitors servers (marked as blue arrows in the figures), and second, when the monitor waits for server push (marked as red arrows.)

Profile Management This component is responsible for registering client or server profiles in the proxy profile base (PB). The profiles are then parsed and validated against the $\mathcal{P}ro\mathcal{M}o$ profile language specification. Resources referenced in the profile domain are extracted and saved in the `Resource Metadata` knowledge base. Notification rules are classified as client requirements and server capabilities according to the profile owner and are then further aggregated and optimized (*e.g.*, using requirement coverage [3]). This component also contains the `Client Notifier` module which is responsible for locating clients that need to be notified, when a notification is due.

Model Management This component contains two sub components that run in parallel, the `Tracker` and `Modeler`. Both run in the background and together are responsible for keeping the resources metadata knowledge base up to date. The `Tracker` tracks resources in the metadata knowledge base and creates a history, i.e., a log of update events occurring at the server. The `Modeler` uses different modeling techniques to generate an update model for resources. Such a model is usually given in stochastic terms. For example, in [4] the use of an update model based on nonhomogeneous Poisson processes was proposed, capturing time-varying update intensities. We shall use this update model in the demonstration. Such a model better reflects scenarios in which RSS feeds are updated more rapidly during work hours, or when eBay bid arrival rate increases towards the end of an auction.

Schedule Management Given client requirements expressed as a set of notification rules over a set of resources in

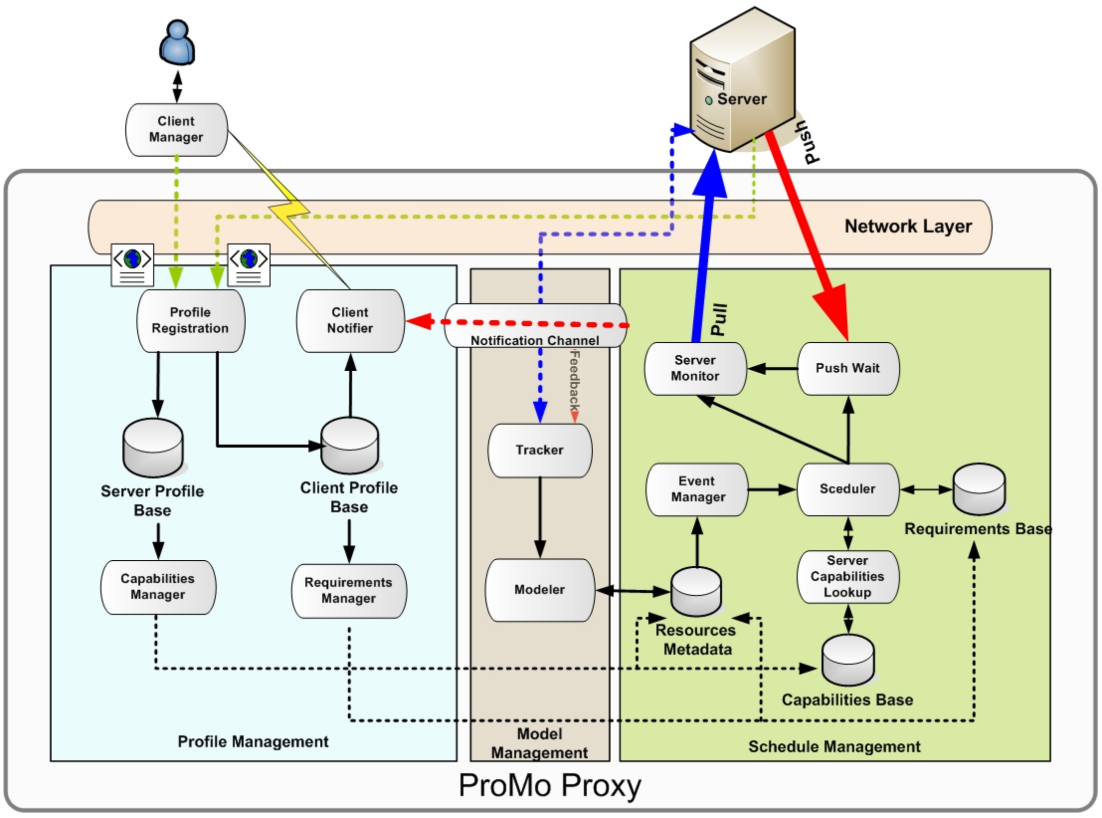


Figure 3: ProMo Architecture

the profile domain, the $\mathcal{P}ro\mathcal{M}o$ scheduler goes through the following process:

- The scheduler reacts to update events generated by the `Event Manager`; this is based on resource metadata and interaction with the `Modeler`.
- On an update event, the scheduler identifies a set of server capabilities that best covers the client notification rule. When such cover cannot be achieved, the $\mathcal{P}ro\mathcal{M}o$ scheduler augments existing server capabilities with *server monitoring tasks* to guarantee satisfaction of client requirements.
- The scheduler then delivers the schedule plan, where *push* actions are assigned to the `Push Wait` module that awaits server push, while *pull* actions are assigned to the `Server Monitor` module to decide on the best plan for server monitoring. This module utilizes the *SUP* algorithm [9] and also its adaptive version, *fbSUP*, to minimize the monitoring tasks while maximizing client utility.
- The collected data is then delivered through the `Notification Channel` module back to the `Profile Management` Component. Data delivered via the notification channel is also used by the `Model Management` component as feedback data for updating the resources metadata.

4. $\mathcal{PROMO}$ DEMONSTRATION DETAILS

Given a profile, server and its capabilities, and a stream of update events, a notification is said to be *executable* when the event specified in its trigger part has occurred and the condition evaluates to true. This interval where a notification is executable is an *Execution Interval*. In this interval, the notification query should be executed and the result delivered to the client. The abstraction of *Execution Interval* serves as a key tool in $\mathcal{P}ro\mathcal{M}o$ scheduling and will be illustrated in the demonstration.

The demonstration will use a trace of RSS Feeds. We collected RSS news feeds from several Web sites such as CNN and Yahoo!. We have recorded the events of insertion of new feeds into the RSS files.

Figure 4 illustrates how a schedule is visualized. The first horizontal bar in the visualization represents the events on a timeline (along the epoch), as fired by the `Event Manager`. The second horizontal bar corresponds to server capabilities. The visualizer uses light gray bars to represent execution intervals corresponding to server push. In the example illustrated in Figure 4, the server capability is *push every third update*. The update policy is `overwrite` which means that each update will overwrite the previous update. Thus, we see that the light gray bar representing the notification interval commences after every third update but is only active until the next (fourth) update event occurs.

The third horizontal bar represents client needs. The visualizer uses a dark black bar to represent execution intervals corresponding to monitoring activities. The client profile requires notification *every other update*. The life policy is

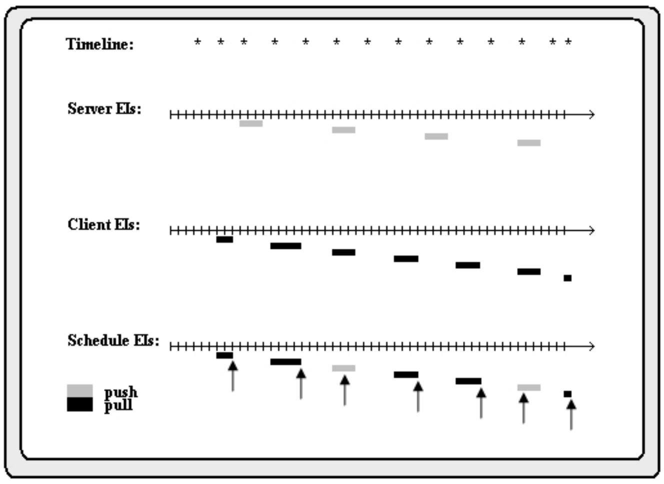


Figure 4: $\mathcal{P}ro\mathcal{M}o$ Schedule Visualization

again `overwrite`, meaning that the client requires to be notified before the subsequence event overwrites the prior event at the server.

The fourth and final bar labeled *Schedule EIs* is the set of execution intervals generated by the $\mathcal{P}ro\mathcal{M}o$ scheduler. It will have a light gray bar when it determines that a client requirement is covered by a server push. It will have a dark black bar when it needs to pull from a server to meet a client requirement. The *push* intervals are assigned to the `Push Wait` module and the *pull* intervals are assign to the `Server Monitor` module for server monitoring tasks. Since the client requirement is *every other update*, and the server capability is *push every third update*, we note that the server capabilities and client needs are not always aligned and that the server push (light gray) execution interval does not cover the client pull execution intervals. It is only at every sixth update event that the server push will meet the client's need. In the other cases, the $\mathcal{P}ro\mathcal{M}o$ schedule has black intervals indicating that the proxy must pull from the server to meet client needs. The black arrows mark the actual times of the *push* or *pull* activities.

Our demonstration will illustrate the results of an optimal *pull* based monitoring algorithm *SUP* (Satisfying User Profiles) and its adaptive version, *fbSUP*; details are in [9]. These algorithms are used by the `Server Monitor` module in $\mathcal{P}ro\mathcal{M}o$ to generate a *pull* schedule which minimizes the monitoring tasks while satisfying client needs.

An alternative method to communicate the effectiveness of $\mathcal{P}ro\mathcal{M}$ scheduling is illustrated in Figure 2. Given an offline event history, $\mathcal{P}ro\mathcal{M}o$ can generate an optimal offline schedule which has maximum utility. Then, using an update model to estimate update events, we can evaluate the $\mathcal{P}ro\mathcal{M}o$ online schedule. A comparison of the online to the offline optimal schedule provides the *Effective Utility* metric.

Figure 5 illustrates the *Effective Utility* for a set of client profiles similar to that in Figure 2. In each profile, we vary the number X, as in *every X updates*; the value of X is plotted on the x axis of Figure 5. The *Effective Utility* in

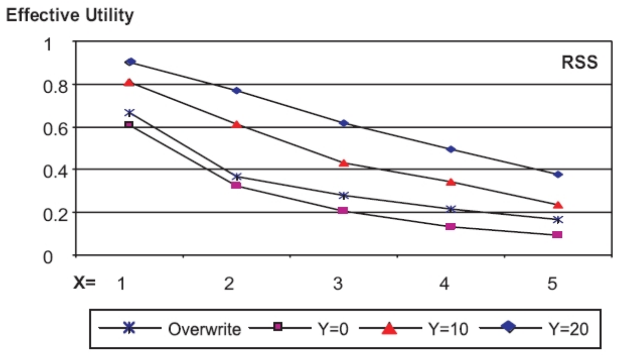


Figure 5: Effective Utility example

the range 0.0 to 1.0 is plotted on the y axis. The four curves represent different *life* parameter settings. This includes the `overwrite` update policy and three values for `Window(Y)`, $Y = 0, 10, 20$. For instance, when `Window(Y=0)`, the client wants to be notified immediately after an update event. As can be seen, as the value of X increases, or for lower values of Y the client profile is more complex or has more constraints. Consequently, the *Effective Utility* will decrease for these parameter settings.

5. REFERENCES

[1] E. Cohen and H. Kaplan. Refreshment policies for Web content caches. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM)*, pages 1398–1406, Anchorage, Alaska, April 2001.

[2] P. Deolasee, A. Katkar, P. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Deisseminating dunamic web data. In *Proceedings of the International World Wide Web Conference (WWW)*, 2001.

[3] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the ACM-SIGMOD conference on Management of Data (SIGMOD)*, page 115126, Santa Barbara, CA, May 2001.

[4] A. Gal and J. Eckstein. Managing periodically updated data in relational databases: A stochastic modeling approach. *Journal of the ACM*, 48(6):1141–1183, 2001.

[5] J. Gwertzman and M. Seltzer. World Wide Web cache consistency. In *Proceedings of the USENIX Annual Technical Conference*, pages 141–152, San Diego, January 1996.

[6] BlackBerry Wireless Handhelds. *http://www.blackberry.com.*

[7] G. Karvounarakis, V. Christophids, S. Alexaki, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *Proceedings of the International World Wide Web Conference (WWW)*, 2002.

[8] Promo language specification. http://ie.technion.ac.il/~avigal/ProMoLang.pdf.

[9] H. Roitman, A. Gal, L. Bright, and L. Raschid. A dual framework and algorithms for targeted data delivery. Technical report, University of Maryland, College Park, 2005. Available from http://hdl.handle.net/1903/3012.

[10] Rss. http://www.rss-specifications.com.