Delft University of Technology
Software Engineering Research Group
Technical Report Series

An Adaptive Push/Pull Algorithm for AJAX Applications

Engin Bozdag and Arie van Deursen

Report TUD-SERG-2008-025

TUDelft

SERG

TUD-SERG-2008-025

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the Third International Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'08)

© copyright 2008, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

An Adaptive Push/Pull Algorithm for AJAX Applications

Engin Bozdag, Arie van Deursen
Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
{v.e.bozdag, Arie.vanDeursen}@tudelft.nl

Abstract

Even though the AJAX *paradigm helps web applications to become more responsive,* AJAX *alone does not provide an efficient mechanism for real-time data delivery. Use cases of applications that need such a service include stock tickers, auction sites or chat rooms. The user interface components of these applications must be kept up-to-date with the latest data from the server, and changes should be received immediately. There are two different static approaches used in the industry to provide real-time data delivery: Either the client pulls for the latest data, or the server pushes it to the client. However, such a static approach is not optimal, since both techniques have their own advantages and disadvantages. In this paper we present an adaptive algorithm that combines both solutions in order to increase scalability, network performance and user-perceived latency.*

1 Introduction

Recently, there has been a shift in the direction of web development. A new breed of web application, dubbed AJAX (Asynchronous JavaScript and XML) is emerging in response to the limited degree of interactivity in large-grain stateless Web interactions. The intent is to make web pages feel more responsive by exchanging small amounts of data with the server behind the scenes and making changes to individual user interface components. This way, the entire web page does not have to be reloaded each time the user makes a change. AJAX is a serious option not only for newly developed applications, but also for existing web sites if their user friendliness is inadequate [14, 13].

The new web applications under the AJAX banner have redefined end users' expectations of what is possible within a Web browser. However, AJAX still suffers from the limitations of the Web's request/response architecture. The classical model of the web called REST [10] requires all communication between the browser and the server to be initiated by the client, i.e., the end user clicks on a button or link and thereby requests a new page from the server. No 'permanent' connection is established between client/server and the server is required to maintain no state information about the clients. This scheme helps scalability [10], but precludes servers from sending asynchronous notifications.

Asymmetry can arise in an environment where newly created items or updates to existing data items must be disseminated to clients. In such cases, there is a natural (asymmetric) flow of data in the downstream direction [1]. There are many use cases where it is important to update the client user interface in response to server-side changes. Examples for such a real-time event notification include an auction web site, a stock ticker, chat application and news portal.

The coherence requirements associated with a data item depends on the nature of the item and user tolerances. For instance, users are generally more tolerant towards latencies in a sports news application than in a stock ticker application. Also, a user who is interested in changes of more than a dollar for a particular stock price need not be notified of smaller intermediate changes [3].

Pull and push are two different approaches in data dissemination (See Section 2 for a more detailed discussion). Simply put, in *pull* the client actively asks for new data continuously, where in *push* the server broadcasts the changes when they are available. Each of these solutions come with their own tradeoffs. Since the data coherence [5] requirements of the users, data publish interval, computation, communication and state-space overheads differ over time, it is very difficult to statically choose between these two solutions.

Because of this, an *adaptive* approach might be a suitable solution for cases where not all key characteristics, such as the data publish interval, are known in advance. With such an adaptive approach, the client or server can *monitor* actual data delivery, latency, or channel update rates, and based on this information adjust the data delivery technique. Thus, a hybrid between pure push and pure pull emerges, capable of supporting more effective and efficient content delivery. In particular, we expect an adaptive approach to achieve

higher availability, network performance, user-perceived latency and scalability.

This paper aims at exploring such an adaptive approach to real time data delivery. The paper reports on our first results, and is structured as follows. Section 2 summarizes the pure push and pure pull approaches. Later, in Section 3, we propose an adaptable solution in which push and pull is combined. Section 4 presents our plan for the evaluation of our algorithm and our testing framework. Finally, Section 6 ends this paper with a conclusion and future work.

2 Real time data delivery techniques

2.1 HTTP Pull

Most AJAX applications check with the server at regular user-definable intervals known as *Time to Refresh* (TTR). This check occurs blindly regardless of whether the state of the applications has changed. The client makes a request to the server, immediately received a response and waits according to TTR (See Figure 1a).

In order to achieve high data accuracy and data freshness, the pulling frequency has to be high. This, in turn, induces high network traffic and possibly unnecessary messages. The application also wastes some time querying for the completion of the event, thereby directly impacting the responsiveness to the user. Ideally, the pulling interval should be equal to the *Publish Rate* (PR), i.e., the rate at which the state changes. If the frequency is too low, the client can miss some updates.

This scheme is frequently used in web systems, since it is robust, simple to implement, allows for offline operation, and scales well to high number of subscribers [11].

Mechanisms such as *Adaptive TTR* [16] allow the server to change the TTR, so that the client can pull on different frequencies, depending on the change rate of the data. Given a user's coherence requirement, this technique allows the server to adaptively vary the TTR value based on the rate of change of the data item. The TTR decreases dynamically when a data item starts changing rapidly and increases when a hot data item becomes cold. Adaptive TTR takes into account (a) a max and min TTR value, so that the TTR value is not set too high or low, (b) the most rapid changes that have occurred so far, (c) the most recent changes to the polled data. In the end, the server calculates the new TTR based on an estimate and the latest TTR values. This dynamic TTR approach in turn provides better results than a static TTR model [16].

2.2 Push

The alternative to pull is push. Figure 1b shows a pure push scheme. In this approach, when the data is available the server opens a connection to the client and pushes the data. This scheme however is not possible to implement in browsers without the installation of a plug-in, since having an open socket on a browser is not possible due to many reasons. To overcome this limitation, a technique called long polling is used in the industry (See Figure 1c). Long polling acknowledges the fact that in order to have a open connection, a request must be coming from the client. However, unlike pull, a response is not immediately returned to the client after a request. The server holds on to the connection until data becomes available. When there is an update to send, it pushes the data to the client, and the client makes another request.

Long polling follows the 'topic-based' [9] publish-subscribe scheme, which groups events according to their topic (name) and map individual topics to distinct communication channels. Participants subscribe to individual topics, which are identified by keywords. Like many modern topic-based engines, it offers a form of *hierarchical addressing*, which permits programmers to organize topics according to containment relationships. It also allows topic names to contain *wildcards*, which offers the possibility to subscribe and publish to several topics whose names match a given set of keywords.

Long polling always guarantees better data coherence comparing to pull and also generates considerably less network traffic [5]. The major problem with this approach however is scalability. There is an open request kept for each client, and this will keep resources busy on the server side. This leads to higher CPU usage [5], and in some servers to server saturation even with a couple of hundred users [4].

Long polling is currently implemented by AJAX libraries such as Dojo[1] and DWR[2]. Since these libraries use the JavaScript engine of the browser, no external plugin download is necessary. This technique is supported by all the major browsers.

3 An adaptive solution

As we discussed in Section 2, both push and pull have their advantages and disadvantages. Most solutions in the industry follow a static approach; either a full pull or a full push mode is used. However, the data coherence requirements associated with a data item depends on the nature of the item and user tolerances. Users will want different data coherence depending on the application, their preferences, bandwidth, etc. Therefore, a hybrid approach will gain the benets of both techniques.

For our solution, we use a modified and extended version of Bhide *et al.*'s Push or Pull (PoP) algorithm [3]. PoP

[1] http://dojotoolkit.org/

[2] http://getahead.ltd.uk/dwr/

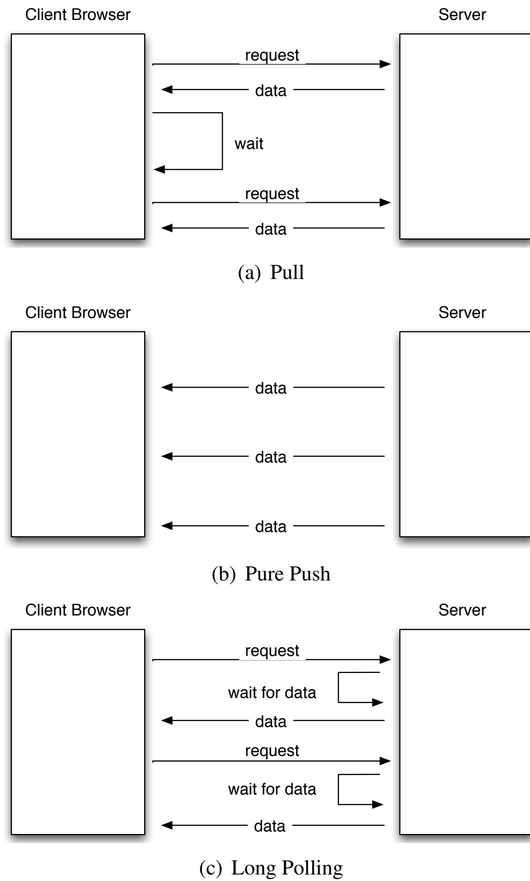

Figure 1. Data Delivery Techniques for AJAX.

is designed to keep proxies up-to-date with the latest data. It does not consider the limitations of browsers and uses a pure push technique, which is not available to browsers. Our algorithm however, is designed for browsers and AJAX applications. In our algorithm users can specify their data coherence requirements using the application. This can also be specified by the application developer, if a single requirement is suitable for all the users of the application.

3.1 Terminology

We use the following terminology in our algorithm:

Publish Triptime: We define trip-time as follows:

Trip-time =
| *Data Creation Date* − *Data Receipt Date* |

Data Creation Date is the date on the publishing server the moment it creates a message, and *Data Receipt Date* is the date on the client the moment it receives the message. Trip-time shows how long it takes for a publish message to reach the client and can be used to find out how fast the client is notified with the latest events.

Data Coherence: We define a piece of data as *coherent*, if the data on the server and the client is synchronized. We check the data coherence of each approach by measuring the trip-time. Accordingly, a data item with a low trip-time leads to a high coherence degree.

3.2 Algorithms

We propose an approach to adaptive push/pull based on three algorithms. First, the *Register* algorithm is used when registering new users that are interested in real time data delivery. Next, there are two *monitoring* algorithms, which keep track of the server and channel performance, and adjust the push/pull settings when necessary. The algorithms themselves are shown in Algorithms 1-3. In the following subsections we discuss these algorithms in detail.

Algorithm 1 Adaptive User Registration

```
1:  procedure REGISTER (User)
2:  if User is already registered then
3:      User := Pull;
4:  else
5:      if Server Load < threshold1 then
6:          User :=Push
7:      else
8:          if Server Load > threshold3 then
9:              repeat
10:                 decreaseServerLoad();
11:             until Server Load < threshold2 OR Timeout
12:         else
13:             if User.Coherence > Default.Coherence then
14:                 User :=Pull;
15:             else
16:                 switchSomePushUsersToPull();
17:             end if
18:         end if
19:     end if
20: end if
21: end procedure
```

Algorithm 2 Monitor Pull Latency

```
1: procedure MONITORPULLLATENCY
2: if User:Pull then
3:     if User.Latency < Max Latency then
4:         Adapt TTR
5:     else
6:         User:=Push;
7:     end if
8: end if
9: end procedure
```

Algorithm 3 Monitor the Update Rate in a Channel

```
1: procedure MONITORCHANNEL
2: if Channel.UpdateRate < Default.UpdateRate then
3:     for Each User in the Channel do
4:         User:=Pull;
5:     end for
6: end if
7: end procedure
```

3.2.1 New User Registration

Procedure *register* in Algorithm 1 shows how a new client is registered. In Lines 1-2, it checks if a client has already registered with the server. This is because having a push user means allocating resources, and to treat clients equally, the same user should not occupy more than one push connection. Also, the number of open XMLHTTPRequest[3] objects on a browser might be limited (usually to 2). If this limit is reached because of having open push connections, this will prevent the browser from making any other requests.

Lines 5-6 check if the server is lightly loaded. It performs this by comparing the current server load by a threshold. If the server is lightly loaded, the new client is automatically registered as a push client. This is because push always provides better data coherence, and since there are enough resources on the server, all clients are provided with high data coherence.

Lines 8-11 check if the server is under maximum load, where the server is almost saturated. If this is the case, the server calls the subroutine *decreaseServerLoad*, which tries to decrease the load. This might be done by:

- Aggregating push data: The server waits for a while before pushing messages, instead of pushing each message separately.
- Filtering: The server filters certain messages and drops them if necessary.
- Delaying: If certain data do not have real-time requirements, they might be delayed.
- Switching: If a user does not have high data coherence requirement, it might be switched to a pull mode, making some resources free.

Lines 13-16 denote the case where the load is critical, but not maximal yet. In this case, if the user's data coherence requirement is not that high (Line 13), it is assigned as a pull client (Line 14). Otherwise (Line 15), the server tries to find some users with low data coherence requirement and switch them to pull, so that there are available resources for this push client. Note that if no resource can be made free, then the client will simply be rejected.

[3] `http://www.w3.org/TR/XMLHttpRequest/`

3.2.2 Monitoring the Latency of Pull Clients

Procedure monitorPullLatency in Algorithm 2 checks the latency of the pull clients in order to adjust the push/pull settings when necessary. In each request, the user also sends the triptime of the previous message. With this information, the server can check if the experienced latency on the client is acceptable (Line 3). If this is the case, in Line 4 the client continues with Adaptive Time To Refresh (TTR). Otherwise (Line 6-7), the user is switched to push, in order to guarantee high data coherence.

3.2.3 Monitoring a Push Channel

It is also quite possible that a push channel is not receiving updates frequently. If this is the case, then unnecessary server resources will be allocated for the subscribers of this channel. In order to optimize the resource usage, channel subscribers can be switched to a pull mode, which will improve scalability. Procedure monitorChannel in Algorithm 3 checks the channel update rate and compares it with a predefined value (Line 2). If the update rate is smaller than this value, then all the push users are switched into pull (Line 3-4).

4 Plan for Evaluation

In order to evaluate the efficiency and performance of our adaptive algorithm, we need to create a distributed testing infrastructure. Unfortunately, distributed systems are inherently more difficult to design, program, and test than sequential systems [2]. They consist of a varying number of processes executing in parallel. A process may also update its variables independently or in response to the actions of another process. Testing distributed programs is a challenging task of great significance. The following problems are of significance in distributed software testing:

- Controllability: This problem occurs when the tester does not know when to send a message, so that a sequence of consecutive transitions cannot be applied in testing [8]. Thus, controllability is the testing power or the capability of the test system to realize input events in a given order [6].
- Observability: Observability may be defined as the testing power or the capability of the test system to determine the output events and the order in which they have taken place [6].
- Reproducibility: We define reproducibility as the ability of a test or experiment to be accurately reproduced, or replicated, by someone else working independently.

Our testing infrastructure must make it possible to control independent variables that are needed in a real-time data delivery application, such as the number of concurrent users and total number of messages, publish interval and pull interval. On top of that, the procedures that we have listed in our adaptive algorithm will create additional variations. The combination of these independent variables and the usage of different algorithms make performing the tests manually an error-prone, and time consuming task.

In order to overcome these challenges, we have created an integrated performance testing framework called CHIRON[4] that automates the whole testing process [5]. As depicted in Figure 2, the controller has direct access to different servers and components (Application server, client simulation server, the statistic server and the Service Provider). By automating each test run, the controller coordinates the whole experiment. This way we can repeat the experiment many times without difficulty and reduce the non-determinism, which is inherent in distributed systems [2]. Since no user input is needed during a test run, observability and controllability problems [8] are minimized.

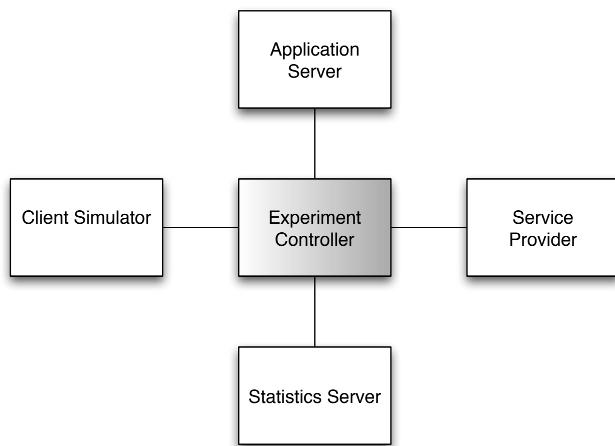

Figure 2. Design of the Distributed Automated Testing Framework CHIRON.

We have implemented CHIRON using a number of open source packages. In particular we use *Grinder*[5] to simulate a big number of virtual users. Grinder seemed to be a good option, providing an internal TCPProxy, allowing to record and replay events sent by the browser. It also provides scripting support, which allows us to create a script that simulates a browser connecting to the push server, subscribing to a particular stock channel and receiving push data continuously. In addition, Grinder has a built-in feature that allows us to create multiple threads of a simulating script. For the application server, we use open-source Jetty 6's Cometd module[6], which is extensible and provided the best results in our previous tests [5]. The statistics server is using Log4j's SocketAppender[7]. For the Service Provider we implemented our own Java application, which uses the HTTPClient library[8]. The controller itself is implemented in Java and uses an open-source SSH library[9] to communicate with different servers. Note that any of these components can be replaced with an alternative solution at future stages. Our steps consist of the following:

- Determine the independent variables, such as number of users, publish interval, pull interval, etc.
- Extend an available push server (Such as Jetty 6) by implementing the adaptive algorithms mentioned in this paper.
- Develop an application that has both push and pull support.
- Implement a script for the Client Simulator with support for switching between push and pull.
- Using CHIRON, establish the initial values of the parameters, such as threshold 1-3 and timeout (Algorithms 1-3).
- Run the tests using CHIRON, obtain and evaluate the performance of the adaptive algorithm by comparing it with push and pure pull.
- Based on the obtained results, tune the parameters of Algorithms 1-3, such as threshold 1-3 and timeout. This will further optimize the results.

5 Related Work

There are a number of papers that discuss a possible adaptive push/pull algorithm. However, most of them focus on client/server distributed systems and non HTTP multimedia streaming or multi-casting with a single publisher.

The research of Acharya *et al.* [1] focuses on finding a balance between push and pull by investigating techniques that can enhance performance and scalability of the system. According to the research, if the server is lightly loaded, pull seems to be the best strategy. In this case, all requests get queued and are serviced much faster than the average latency of publishing. The study is not focused on HTTP.

Bhide *et al.* [3] also try to find a balance between push and pull, and present two dynamic adaptive algorithms:

[4] In Greek mythology, CHIRON, was the only immortal centaur. He became the tutor for a number of heroes, including AJAX.

[5] http://grinder.sourceforge.net

[6] http://www.cometd.com

[7] http://logging.apache.org/log4j

[8] http://hc.apache.org/httpclient-3.x

[9] http://www.trilead.com/Products/Trilead-SSH

Push and Pull (PaP), and *Push or Pull* (PoP). According to their results, both algorithms perform better than pure pull or push approaches. Even though they use HTTP as messaging protocol, they use custom proxies, clients, and servers. They do not address the limitations of browsers nor do they perform load testing with high number of users.

Cao *et al.* [7] discuss push and pull for mobile agent (MA) technology, in which autonomous objects or cluster of objects are able to move between locations. Authors mention the properties and trade-offs of both approaches for mobile agents and relay stations. They present an adaptive pull algorithm to decide the query frequency (Time to Refresh) by considering several factors including the distance between the agent and the relay station. The paper does not address browsers, the web and the AJAX technology.

Liu *et al.* [12] investigate efficient strategies for supporting on-demand information dissemination and gathering in large-scale wireless sensor networks. In the paper, authors try two combine two techniques in order to balance push and pull in data gathering and dissemination for large-scale wireless networks. They show that their comb-needle scheme performs better than pure push or pure pull. The study focuses on reliability and query coverage of sensors in large wireless networks.

Saxena *et al.* [15] introduce a hybrid scheduling algorithm that probabilistically combines the number of push and pull operations depending on the number of items present in the system and their popularity. The access probabilities of the data items are computed dynamically, without any prior knowledge. Authors compare the performance of various push and pull scheduling algorithms and select the scheme providing the best performance. They use a cut-off point to separate the push and the pull sets. The work is focused on wireless networks.

6 Conclusion

In this paper we pointed out the advantages and disadvantages of pull and push solutions for achieving web-based real time event notification. Starting out from these trade-offs, we proposed an adaptive algorithm that combines these two techniques in order to gain the benefits of them both.

Our future work includes possible optimizations, the full implementation of the algorithm and testing it with our Distributed Automated Testing Framework CHIRON[10].

[10] CHIRON is available to download as an open-source project. See http://spci.st.ewi.tudelft.nl/chiron

References

[1] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 183–194. ACM Press, 1997.

[2] S. Alager and S. Venkatsean. Hierarchy in testing distributed programs. In *AADEBUG '93: Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 101–116, London, UK, 1993. Springer-Verlag.

[3] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Trans. Comput.*, 51(6):652–668, 2002.

[4] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of push and pull techniques for Ajax. In S. Uang and M. D. Penta, editors, *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE)*, pages 15–22. IEEE Computer Society, 2007.

[5] E. Bozdag, A. Mesbah, and A. van Deursen. Performance testing of data delivery techniques for Ajax applications. Technical report, Delft University of Technology, 2008.

[6] L. Cacciari and O. Rafiq. Controllability and observability in distributed testing. *Information & Software Technology*, 41(11-12):767–780, 1999.

[7] J. Cao, X. Feng, J. Lu, H. C. B. Chan, and S. K. Das. Reliable message delivery for mobile agents: push or pull? *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 34(5):577–587, 2004.

[8] J. Chen, R. M. Hierons, and H. Ural. Overcoming observability problems in distributed test architectures. *Inf. Process. Lett.*, 98(5):177–182, 2006.

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

[10] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.

[11] M. Hauswirth and M. Jazayeri. A component and communication model for push systems. In *ESEC/FSE '99*, pages 20–38. Springer-Verlag, 1999.

[12] X. Liu, Q. Huang, and Y. Zhang. Combs, needles, haystacks: balancing push and pull for discovery in large-scale sensor networks. In J. A. Stankovic, A. Arora, and R. Govindan, editors, *Proc. of the 2nd Int. Conference on Embedded Networked Sensor Systems, SenSys 2004, Baltimore, MD, USA, November 3-5, 2004*, pages 122–133. ACM, 2004.

[13] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In D. Schwabe and F. Curbera, editors, *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*. IEEE Computer Society, July 2008.

[14] A. Mesbah and A. van Deursen. A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software (JSS)*, 2008. Accepted for publication.

[15] N. Saxena and M. C. Pinotti. Performance guarantee in a new hybrid push-pull scheduling algorithm. In Q. H. Mahmoud and H. Weghorn, editors, *Wireless Information Systems, Proc. of the 3rd Int. Workshop on Wireless Information Systems, WIS 2004, In conjunction with ICEIS 2004, Porto, Portugal, April 2004*, pages 50–62. INSTICC Press, 2004.

[16] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual data warehouses. In *IEEE Real-Time Systems Symposium*, page 60, 1998.

TUD-SERG-2008-025
ISSN 1872-5392