Adaptive memory-aware chunk sizing techniques for data-intensive queries over Web Services

Anastasia Theodouli
Dept. of Informatics
Aristotle University of Thessaloniki, Greece
anastath@csd.auth.gr

Anastasios Gounaris
Dept. of Informatics
Aristotle University of Thessaloniki, Greece
gounaria@csd.auth.gr

ABSTRACT

Modern applications of Web Services (WSs) that involve the processing of large amounts of data tend to transmit data in chunks. Several performance control techniques have been proposed to dynamically select the appropriate chunk size with a view to minimize the communication cost. However, when the data consumer is slower than the data producer, the consumer applications may suffer from memory shortage if high volumes of data arrive in the incoming buffers. To this end, we propose a specific approach to coupling performance control with congestion control features, in order to consider both performance and memory overflow issues in an integrated manner. The performance results with real data show that we can combine these controllers effectively and efficiently, so that no memory overflow occurs at the expense of negligible performance degradation.

1. INTRODUCTION

Web services (WSs) allow the development of network-available applications in a standard, rapid, easy and low-cost way; their proliferation has imposed a profound impact on both the software development practices, giving rise to the service-oriented architecture paradigm, and the way enterprize and scientific applications are integrated and exposed over the Internet [7]. Currently, the example usages of WSs cover a particularly wide spectrum of applications, which are compatible with modern web, grid and cloud infrastructures.

This work has been largely motivated by applications of WSs that involve the processing of large amounts of data. Several WS-based scientific applications fit into this category (e.g., [6, 5]). A common characteristic in such scenarios is that WSs are typically repeatedly accessed and the volume of data to be transferred remotely and processed by the WSs is high. A similar setting is the one provided by application-generic service-based query processing tools, which assume a dataflow that is processed with the help of query execution plans comprising accesses to WSs (e.g., [1, 10, 9]).

Without loss of generality, we assume a setting that is very common in OGSA-DAI applications. OGSA-DAI is a pioneer system in WS-based query processing and provides a framework for access and management of remote and distributed databases that are exposed as WSs; OGSA-DAI has been successfully applied to several domains, including astronomy, meteorology, medical research, computer-aided design and engineering [1]. In Fig. 1, we depict the main interactions that involve passing messages over the network. The client program consists of two threads, namely a thread for transferring the data from the remote source, and a thread that performs the actual processing. In *Step 1*, the client program issues a query, which, in the cases considered in this work, returns a high volume of data for further processing on the client side. The service-wrapped database receives the query and prepares the results. Typically, the results are not sent in a single message, but are split into chunks [10, 3]. More specifically, in *Step 2a*, the client asks for the next chunk of the results specifying the exact chunk size (measured in number of tuples) as a parameter. Upon receipt of such a request, the WS encapsulating the database, sends a chunk of results (*Step 2b*), which are stored in an input queue. Steps 2a and 2b are repeated as many times as needed to retrieve the complete result set. In parallel with the data transfer thread, the data processing thread retrieves data items from the input queue in order to process them.

Problem Description. As mentioned above, the fact that the data volume to be transferred is split into chunks introduces a new problem, namely the selection of the appropriate chunk size. In [10, 3], firstly, it has been reported that the size of the chunk size affects the overall performance and the cost to transfer a dataset remotely, and secondly, appropriate performance control techniques to minimize this cost have been proposed. These techniques decide the optimal size of the chunks, either in a static or in an adaptive manner. In general, the cost per data item is a unimodal function of the chunk size: for small chunk sizes the cost is high and decreases for larger chunk sizes up to a point where further

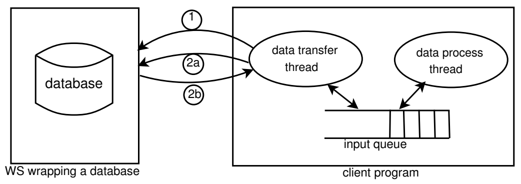

Figure 1: High-level description of our setting.

increases in the chunk size lead to increase in the cost as well; the techniques in [10, 3] basically aim to efficiently detect that point. Although such techniques are appropriate when the cost to transmit a chunked dataset dominates, they fall short when the network is fast relatively to the speed that the consumer service processes data. In those cases, data is accumulated in the WS incoming queue because the data transfer thread is faster than the data processing thread, and thus, after a point, memory overflow occurs, which, in turn, leads to significant performance degradation, e.g., due to thrashing, or even WS failure.

Our contribution lies in that we extend the existing adaptive techniques for chunk size selection, so that applications are endowed with the capability to take into account both the need to transmit data as quickly as possible, and the need to keep the volume of buffered data below the size of the memory that is available. More specifically, we propose to integrate congestion control with performance control when tuning the chunk size at runtime, and, to this end, we draw inspirations from lower-level TCP buffer sizing techniques. The performance results with real data show that we can combine these controllers effectively and efficiently, so that no memory overflow occurs at the expense of negligible performance degradation. Our controller operates at the client side and on top of any existing network protocols (e.g., SOAP/HTTP for the OGSA-DAI setting). In other words, we do not propose any new application protocol, but appropriate techniques to specify the requested chunk size in Step 2a in the setting of Fig. 1[1].

Note that the novel technique that we propose can be extended to non-WS scenarios, since it is applicable to any setting where i) data is retrieved by a processing client in chunks of configurable size and ii) the data consumer is slower than the data producer (at least on average). Such cases may be encountered in more traditional data management; e.g., the work in [2] provides evidence about potential memory overflows when employing the well known pipelined join algorithm in [12]. Other cases that are tailored to our technique are those where the data consumer performs an intensive task, such as on-the-fly outlier detection.

Structure. The remainder of this article is structured as follows. We discuss related work in Sec. 2. Our extensions regarding congestion control, which come in two flavors, are thoroughly presented in Sec. 3. Our proposal is evaluated using real data, and the evaluation results appear in Sec. 4. Finally, Sec. 5 concludes the paper.

2. RELATED WORK

The related work on data chunk sizing can be classified in two main categories depending on the application setting. The first category refers to the scenarios that deal with chunk sizing during data transmission over remote software services mainly with a view to improving the performance. The second category regards a similar problem at a much lower network level, and deals with the chunk sizing problem in the context of the TCP protocol mainly with a view to avoiding memory overflow.

Data chunk sizing for software/WS clients. Transferring large amounts of data residing on distributed and heterogeneous databases which are exposed via WSs has posed an urge for implementing several techniques of chunked data transmission. This urge derives mostly from the fact that WSs are slow and thus the communication cost per se becomes the bottleneck. As mentioned before, the per data item cost of a chunk follows a unimodal function of the chunk size in the generic case, and as such there is a global minimum. Also, typically, there are minimum and maximum limits in the chunk size (e.g., imposed by the network packet lengths). The work in [10] presents a static approach that defines the optimal chunk size through profiling information.

The techniques in [3] deal with an adaptive version of the problem of deciding the optimal chunk size, which is tailored for dynamic, volatile environments with no a-priori profiling information. They assume a client that pulls data from a remote store in chunks (as we do) and they are based on a switching extremum control algorithm that optimizes the data transfer by tuning the data chunk to be transferred in each step taking into account the average transmission time of the previous data chunks. As such, this solution is based on control theory and more particular in the feedback (or closed) loop control systems. Our work is orthogonal to those techniques, since we focus on congestion control; moreover, our proposal can encapsulate any performance controllers as explained in Sec. 3.

Data chunk sizing in TCP. The notion of data partitioning in an attempt to control the flow of data transmitted between two peers is also introduced in the TCP protocol. Both in TCP Flow and Congestion control, the amount of data to be transferred between the two peers, termed as sender and receiver, is predefined by taking into account the receiver's available buffer size and the network congestion, respectively. The *Sliding Windows* technique [11, 8] used in TCP Flow Control aims at avoiding the throttling of the receiver due to a faster sender. The upcoming network congestion can also be detected by the TCP intermediate routers that apply techniques such as *Active Queue Management (AQM)* in order to notify the TCP endpoints about early congestion; one technique that leverages AQM is *Random Early Detection (RED)* [4]. These techniques have inspired our work; however our congestion control operates at a higher level than TCP. In general, chunk sizing has been employed in network buffers to achieve high link utilization. Real links almost always contain a complex mix of flow connection lengths, round-trip times, UDP traffic and so on, and measurements conform that traffic patterns change significantly over time; this motivates adaptive approaches to buffer sizing, where attention must be paid to cases where no data loss is allowed [13].

3. MEMORY CONGESTION CONTROL

Since most WSs exhibit poor performance, e.g., due to parsing and data transfer overheads [10], a major concern in a large portion of WS-based applications is the data transmission cost. To this end, the algorithms that we presented in the previous section have been proposed in order to cope with the data transmission cost minimization. Nevertheless, there is an increasing number of WS-based applications that are not network-bounded. In such applications, the throughput depends mostly on the data processing capability of the data consumer. More specifically, whenever the client processes the data at an average rate that is slower than the rate it receives it, data gets accumulated in the client's input queue; as such, it is possible to experience memory conges-

[1]Our chunk sizing technique should not be confused with the HTTP chunked transfer encoding.

tion at the client side. If memory congestion persists and is not addressed, it may lead to memory overflow with detrimental impacts on the performance. Note that applications that employ a performance controller are more prone to this phenomenon, due to their capability of optimizing the transmission from the data source to the data consumer.

A naive solution to the problem of memory congestion would be to modify the client logic, so that i) the data is processed as soon as it arrives at the consumer side; and ii) the next chunk of data is requested only after the processing of the previous chunk has finished. Obviously, in such a scenario, the maximum amount of data that is temporarily stored in the incoming buffers of the client is equal to the size of biggest chunk requested. If that size is no greater than the memory available, then no memory overflow can occur. However, such a solution is unacceptably slow, since the data communication and processing activities take place in a sequential rather than in a parallel manner.

In this work, we have developed a solution that does not compromise the overlapping between processing and communication time, and combines performance control with memory congestion control thus resulting in an integrated controller to run at the client side; such a controller adjusts the chunk size requested at each step so that the transmission cost is minimized while no memory overflow can happen. In fact, we introduce an application of a two-layer control, where congestion control is activated when memory congestion is detected. Our memory congestion control proposal is inspired by and based on some of the TCP/IP control techniques of the Internet protocol suite lower level layers (discussed in Sec. 2.). However, our controller operates at a much higher level, regards the internal behavior of the communication links as a black box, and is independent of any network transport protocol implementation.

3.1 The two flavors of the controller

We present two controllers for memory congestion: the first one builds upon ideas from TCP Flow Control and Active Queue Management (AQM) [4], whereas the second is inspired by the TCP/IP congestion handling mechanisms. These controllers take over, when the incoming memory buffers are running out of free space, as explained later.

Flavor I: Memory congestion control based on Active Queue Management and TCP/IP flow control. On the application layer, the client-side controller uses a control technique that is used both in TCP/IP flow control and in AQM. In both these mechanisms, the next data chunk size to be transferred is decided based on the available buffer size at the side of the receiver when referring to the TCP/IP flow control and of the bottleneck router when referring to the AQM. As such, the way memory congestion is handled depends on the intensity of the congestion.

The details are as follows. During TCP flow control, the TCP receiver of an established TCP connection, which corresponds to our data consumer that pulls data from the data source, advertises to the TCP sender the size of the next packet (the so called *window* or *rwnd*) that is willing to accept in order to avoid the congestion on its input buffer. The receiver's advertised window is calculated by the equation:

$$Window = rcvBuffer - (lastByteRcvd - lastByteRead)$$

where *Window* is the *rwnd*, *rcvBuffer* is the size of the receiver's buffer, *lastByteRcvd* is the sequence number of the last byte that was received, and *lastByteRead* is the sequence number of the last byte that was read and thus removed from the input buffer.

In the equation above, it can be easily observed that the window corresponds to the available input buffer size of the receiver. The AQM mechanism relies on a similar rationale: In the context of an established connection, the intermediate routers constantly check the size of the used input buffer size and of the available space in order to detect early congestion.

Our controller leverages the afore-mentioned rationale, as it cannot adopt it in a straightforward manner. The reason is that our controller is located at the client side and has full control of the next chunk size to be requested; this holds true for any pull-based data transmission. The consequence is that simply advertising the available buffer size does not yield any results in our case. So, we propose an approach according to which, in each step, we check the available size of the input buffer. Then we calculate the size of next chunk as a percentage of the available buffer size in the following way:

$$chunkSize = \beta \cdot availableBufferSize \tag{1}$$

where β is a configurable parameter. Through experimentation we have found out that setting β to 0.1 is a good value that helps in avoiding memory congestion, although our algorithm is rather insensitive to different β values. More results along with sensitivity analysis for this parameter appear in the evaluation section.

Flavor II: Memory congestion control based on TCP/IP congestion control. The second flavor builds on top of techniques used by the TCP/IP Congestion Control. More specifically, in one of the algorithms used in TCP/IP congestion control, called *fast recovery*, the window is reduced to the half of its previous value when the sender receives the same acknowledgement (ACK) three times, which is an evidence of congestion; after this reduction, the sender increases the window linearly in accordance with the congestion avoidance algorithm and uses ACKs received by the receiver in order to determine the rate of packet transmission.

Our controller imitates the aforementioned rationale by reducing the next chunk size to the half when it detects memory congestion at the client-side. The equation used to calculate the next chunk size appears below, where it is assumed that memory congestion has been detected during the k-th step:

$$chunksize_{k+1} = \frac{chunksize_k}{2} \tag{2}$$

The relative performance of the two flavors is investigated and discussed in Sec. 4.

3.2 An integrated controller

In order to benefit from both the performance and the congestion control, we proceeded in the integration of the two controllers thus resulting in a hybrid two-layer controller that operates as a performance controller when no memory congestion is detected, and switches to a memory congestion controller if the memory available is lower than a threshold. Note, that there are multiple flavors of performance control (e.g., several flavors of a performance controller are discussed in [3]). Any of such flavors can be combined with both the flavors of congestion control presented above, which gives rise to a big number of variants of the integrated controller.

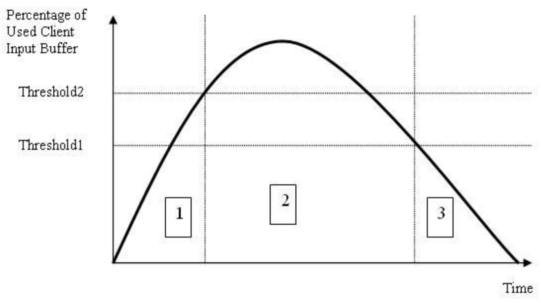

Figure 2: The control areas in the integrated controller (threshold1=lt; threshold2=ut).

Central to our integrated controller, is the detection of memory congestion. This is performed with the help of a configurable threshold *ut*, which denotes the upper threshold of the extent to which the memory is full; a typical value of *ut* is 80%. When the memory available is less than *1-ut*, then the congestion control becomes responsible for the decisions on the size of the next chunk. However, we do not allow the performance controller to resume as soon as the occupied memory decreases below *ut* as a result of the congestion control. The reason is that such an approach is particularly prone to instability. To avoid that problem, we re-activate the performance controller only after the occupied memory becomes lower than a second threshold, *lt* (lower threshold), where $ut > lt$. A typical value of *lt* is 60%.

Through the incorporation of the two thresholds mentioned above, the operating area of the integrated controller is partitioned in three sub-areas, as shown in Fig. 2. The first area corresponds to the case that no memory congestion has been detected and the size of the next chunk is decided by the performance controller. Without loss of generality, we assume that, as time goes by, data is accumulated to the incoming buffer, so that, after a time point, the size of this data exceeds *ut*. Then we enter, the second sub-area, where the congestion control is activated. The role of the congestion controller is not to allow the buffer to become completely full, which results in memory overflow, and the main mechanism to this end is to reduce the size of the chunks requested as explained earlier. The congestion controller is de-activated, when the amount of temporary data becomes lower than *lt* times the size of the incoming buffer. That corresponds to the third area, in which the performance controller defines the chunk size. Note that the pattern depicted in Fig. 2 may be repeated arbitrary times during the transmission of a large dataset.

The pseudocode of the integrated controller is in Algorithm 1, where we present the version of the integrated controller that implements the first flavor of the congestion controller. The *PerformanceControl* function encapsulates any performance controller that we may employ, as discussed above. The algorithm also employs a variable *rising* in order to detect the operating area, and thus the type of the controller to be applied. Note also that, when memory congestion is detected, no chunk is requested for a *sleeptime* period (lines 18 and 24); this heuristic further helps in avoiding memory overflow. For the second flavor of the congestion control, the changes in the algorithm are trivial: lines 20 and 26 are simply replaced with Equation (2). In all cases, the time complexity for computing the next chunk size is $O(1)$.

Algorithm 1 Integrated Controller

```
1:  $chunkSize \leftarrow initialChunkSize$
2:  $congestionDetected \leftarrow$ **false**
3:  **while** $endOfResults$ $is$ **false do**
4:     **if** $occupiedBufferSize < lt$ **then**
5:        $rising \leftarrow$ **true**
6:        $congestionDetected \leftarrow$ **false**
7:        $t_1 \leftarrow timestamp()$
8:        $WebService.requestNewBlock(chunkSize)$
9:        $t_2 \leftarrow timestamp()$
10:       $chunkSize \leftarrow PerformanceControl(t_2 - t_1, chunkSize)$
11:    **else if** $occupiedBufferSize < ut$ **then**
12:       **if** rising is **true then**
13:          $t_1 \leftarrow timestamp()$
14:          $WebService.requestNewBlock(chunkSize)$
15:          $t_2 \leftarrow timestamp()$
16:          $chunkSize \leftarrow PerformanceControl(t_2 - t_1, chunksize)$
17:       **else**
18:          $WebService.sleep(sleeptime)$
19:          $WebService.requestNewBlock(chunkSize)$
20:          $chunkSize \leftarrow \beta \cdot freeBufferSize()$
21:          $congestionDetected \leftarrow$ **true**
22:       **end if**
23:    **else**
24:       $WebService.sleep(sleeptime)$
25:       $WebService.requestNewBlock(chunkSize)$
26:       $chunkSize \leftarrow \beta \cdot freeBufferSize()$
27:       $rising \leftarrow$ **false**
28:       $congestionDetected \leftarrow$ **true**
29:    **end if**
30: **end while**
```

4. EVALUATION

The evaluation of our approach is based on real data with regards to the actual transmission time of chunks of data. We installed an OGSA-DAI service, which provides a service interface to pull data from a background database in chunks of configurable size. As explained earlier, our approach is tailored to environments, where the communication cost is not the only dominant cost; more specifically, we are interested in case where the average computation cost is at least as high as the average communication cost at least for certain chunk sizes, so that memory congestion phenomena may arise. In our experiment setting, we used a client that is connected to the same LAN as the database server. In all experiments, the chunk size is measured in the amount of database tuples retrieved; each tuple has an average length of approximately 100 bytes thus the size of each chunk in bytes can be computed in a straightforward manner as well.

In the following, we first present the profiles of the function of the per tuple communication cost as a function of the chunk size, and then we present our experiments. The aim of the experiments is: i) to demonstrate the effectiveness of our congestion controllers to avoid memory overflow issues; (ii) to show that the aforementioned effectiveness is combined with high efficiency, in the sense that the performance degradation when no memory congestion occurs is negligible; and (iii) to provide evidence about the sensitivity of our results with regards to the controller parameters

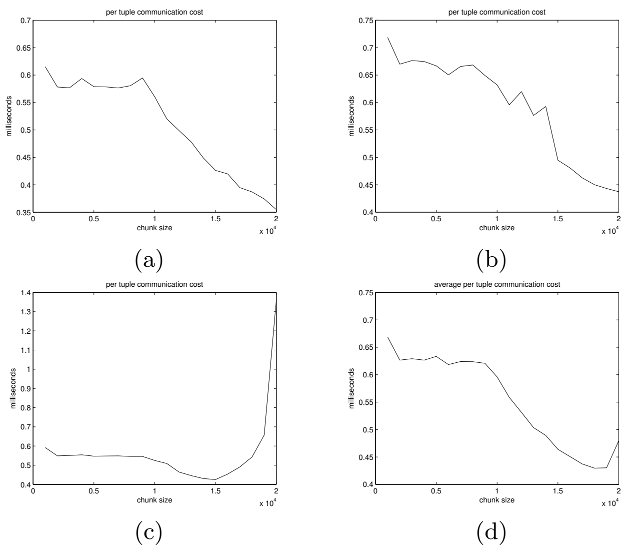

Figure 3: (a)-(c): profiles of the per tuple communication cost in different time periods; (d): the average per tuple communication cost.

and the evaluation environment settings.

Profiles. In order to construct the real profiles of the per tuple communication cost for different chunk sizes, we took measurements in 12 distinct time periods, during which both the client and server machines were relatively unloaded (i.e., they were running only the client and the database and web server software, respectively), but the network was shared among dozens of users. In each time period, we took measurement for chunk sizes between 1000 and 20000 tuples. For each chunk size, we repeated the measurements 10 times, and we created a profile for each time period; i.e., overall we created 12 profiles. Fig. 3 shows some representative samples. In Fig. 3(a) and (b), the profiles contain numerous local minima, but in general the per tuple communication cost decreases with increased chunk sizes. The point that the decrease becomes steep and the features of the local optimal areas differ. Fig. 3(c) represents a profile that is more typically encountered in wide settings, given also the evidence in [3]. In our experiments, such a profile trend appeared only once in the 12 time periods and provides evidence that allowing the chunk size to increase arbitrarily may lead to detrimental effects. Finally, the average behavior is depicted in Fig. 3(d).

4.1 Experiments

The experiments to evaluate our proposal utilize the profiles constructed in the way described above so that a realistic environment is emulated; this is done in order to avoid non-determinism in the experiments while taking measurements that fully map to realistic settings. So, in our experiments, the client adopts a modified version of the Algorithm 1, where there is no actual data transmission, but the client calculated the time cost need to receive a chunk size based on the profiles. The second modification we made to the algorithm is that we imposed hard limits on the smaller and bigger chunk size, as discussed earlier.

The evaluation environment is parallel: we assume that the data consumer on the client side, processes data as soon as it arrives in parallel with the arrival of additional data chunks. As such, it constantly checks its internal buffer where data arrives to check whether it is empty or not; and if it is not empty it processes the next tuple. Regarding the performance controller, we take the simplest approach that is compliant with the profiles constructed: we assume that the chunk size is fixed at 20000 tuples, which is the value that leads to the optimal per tuple communication cost in 11 out of 12 profiles. More sophisticated dynamic solutions can converge to this value, even if that was not known a-priori. In other words, our selection for this constant chunk size is equivalent to a the choice of an effective performance controller while avoiding any problems stemming from non-determinism in our experiments. This allows us to concentrate solely on the impact of the congestion controller.

Parameter	Value
Result Set	1000000 tuples
initialChunkSize	20000 tuples
sleeptime	10 msecs
β (used in flavor I)	0.1
lt	0.6
ut	0.8

Table 1: Fixed parameters for the first set of the experiments

per tuple processing cost (in msecs) - memory available	0.5	0.6	0.7	dynamic	aggregate
5 %	0/0/11	2/2/11	3/3/11	0/0/11	5/5/44
25 %	0/0/3	0/0/11	3/3/11	0/0/11	3/3/36
50 %	0/0/0	0/0/0	0/0/3	0/0/0	0/0/3
75 %	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
aggregate	0/0/14	2/2/22	6/6/35	0/0/22	**8/8/83**

Table 2: Number of the occurrence of memory overflows for flavor I / flavor II / No congestion control.

Avoidance of memory overflow. Initially, we examine the capability to avoid memory overflow. We experiment with i) different sizes of the memory available; and ii) different speed of processing of incoming tuples (so that the space in the memory buffer is freed). The rest of the parameters and the environmental settings are shown in Table 1.

Table 2 shows the number of memory overflow for the two flavors of the memory congestion controller compared to the case that there is no chunk sizing control and the chunk size remains fixed throughout the data transmission. The values of the available memory we examined are 5%, 25%, 50% and 75% of the complete dataset to be transmitted. The values for the processing speed per tuple are chosen based on the profiles so that they both are commensurate with the transmission cost per tuple and may lead to memory congestion. More specifically, we experimented with 3 fixed consumer speeds (0.5, 0.6, and 0.7 msecs) and a dynamic one, where, for each profile, the consumer speed was equal to the per tuple transmission cost when the chunk size is 1000. When the value of the data consumer speed is 0.5 msecs, then the data processor is relatively fast, whereas, when the value is 0.7 msecs, the consumer is relatively slow.

In each cell of Table 2, we show for how many of the 12 profiles, memory overflow took place. For example, when the consumer speed is fixed at 0.7 for all profiles, and the memory available can store only 5% of the complete result set of 1000000 tuples, then memory overflow happens 3 times if we employ congestion control and 11 times if we do not employ congestion control. As expected, the number of memory overflow decreases as the memory available increases and/or the consumer processing cost per tuple decreases. On average we examined $4 \cdot 4 \cdot 12 = 192$ profile-configuration pairs. Without memory congestion control, in 83 cases we experience memory overflow, whereas, with the help of memory congestion control, the number of time this happens is 8 only; the cases that memory overflow takes place despite the existence of a congestion controller refer to scenarios where the memory available is small and the consumer too slow for the congestion controller to avoid overflow by asking smaller chunks. Note that the effects of the two flavors are similar. This proves the effectiveness of our approach to avoiding memory congestion.

Performance Overhead. A question may arise as to whether there is a trade-off between tolerance to memory congestion problems and performance. To investigate this hypothesis, we compared the aggregate running time of both the data transmission and the processing task for all the configurations, for which all controllers did not exhibit memory overflow. In total $192 - 83 = 109$ such configurations meet these criteria. The evaluation results show that the difference between the running time is within a range of $\pm 0.01\%$. Consequently, we can claim that the associated overhead (i.e., the time cost of the incurred delays) when there is no need for memory congestion control is totally negligible.

Sensitivity analysis. We have experimented with a wide range of values for the *sleeptime* and β parameters, and the lt and ut thresholds. No detailed figures are presented due to space constraints. More specifically, for a fixed value of memory available (25%) and consumer processing rate 0.6 msecs or 0.7 msecs, the sensitivity analysis results are as follows. Varying the β parameter of the first flavor from 0.001 to 0.3 and keeping the rest of the parameters as they appear in Table 1, did not lead to a modification in the number of memory overflows. However, the *sleeptime* parameter had a (positive) impact on the effectiveness. For consumer processing rate at 0.7 msecs per tuple, flavor I exhibited 2 memory overflows if the *sleeptime* increased to 100, and no overflows at all if it increased to even larger values (e.g., 500). Flavor II did not experience any memory overflow even for *sleeptime* set to 100. The values of the threshold tested were between 0.3 and 0.7 for lt, and 0.7 and 0.95 for ut. The only observed impact on the effectiveness of the congestion controllers compared to the values in Table 2 was 2 memory overflows when the processing rate is 0.6 msecs and the ut is 0.95. Overall, the analysis shows that the congestion controllers are rather insensitive to their parameters and can totally eliminate memory overflows if *sleeptime* is increased significantly. Also, none of the values considered had resulted in a significant increase in the running time in cases where there was no memory congestion.

Comparison of the two flavors. The two flavors exhibit similar performance as shown in the results presented above. As such, there seems to be no clear winner between them. However, the first flavor has one input parameter more, namely the β parameter, and thus may be slightly less practical in some applications. Nevertheless, this is not supported by the sensitivity analysis above, where both flavors have been shown to be rather insensitive to their parameters.

5. CONCLUSIONS

Nowadays, more and more data sources are publicly available on the web and may be exposed as services. Transferring large datasets to remote data consumers may incur a high communication cost that is mitigated by splitting the datasets in several chunks. The problem we deal with is that, when a remote data source sends chunks of data at a speed higher than the processing speed of the data consumer, the data consumer may experience memory congestion problems or even memory overflow if the amount of available memory is limited. In this paper, we present a two-flavored solution to this problem. More specifically, we propose an integrated controller that dynamically tunes the chunk size at runtime so that both the communication cost is minimized and no memory overflow occurs. Our memory congestion control mechanisms have been inspired by their counterparts incorporated within the TCP/IP protocol; however, in our case we have managed to efficiently apply those techniques at a higher level where a data consumer interacts with a service-based database using SOAP/HTTP. The evaluation was based on real data. The results presented support our claim that our proposal is effective, in the sense that it reduces the cases where memory overflow occurs, and efficient, in the sense that it incurs negligible overhead. An interesting avenue for future work is to investigate the impact of choices regarding lower-level network layers on our approach.

6. REFERENCES

[1] M. Antonioletti *et al.* The design and implementation of grid database services in OGSA-DAI. *Concurrency - Practice and Experience*, 17(2-4):357–376, 2005.

[2] M. A. Bornea, V. Vassalos, Y. Kotidis, and A. Deligiannakis. Adaptive join operators for result rate optimization on streaming inputs. *IEEE Trans. Knowl. Data Eng.*, 22(8):1110–1125, 2010.

[3] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. D. Dikaiakos. Robust runtime optimization of data transfer in queries over web services. In *Proc. of ICDE*, pages 596–605, 2008.

[4] C. Hollot, V. Misra, D. Towsley, and W. Gong. Analysis and design of controllers for aqm routers supporting tcp flows. In *IEEE Trans. on Automatic Control*, volume 47, June 2002.

[5] D. T. Liu and M. J. Franklin. The design of griddb: A data-centric overlay for the scientific grid. In *VLDB*, pages 600–611, 2004.

[6] S. Narayanan, U. V. Catalyrek, T. M. Kurc, X. Zhang, and J. H. Saltz. Applying database support for large scale data driven science in distributed environments. In *GRID*, 2003.

[7] M. Papazoglou. *Web Services: Principles and Technology.* Pearson, 2nd edition, 2011.

[8] Peterson and Davie. *Computer Networks A Systems Approach.* Morgan Kaufmann, 3rd edition, 2003.

[9] M. Sabesan and T. Risch. Adaptive parallelization of queries over dependent web service calls. In *WISS, ICDE*, pages 1725–1732, 2009.

[10] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB*, pages 355–366, 2006.

[11] W. Stallings. *Data and Computer Communications.* Prentice-Hall, 6th edition, 2000.

[12] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. Rpj: Producing fast join results on streams through rate-based optimization. In *SIGMOD*, pages 371–382, 2005.

[13] G. Vu-Brugier, R. S. Stanojevic, D. J. Leith, and R. N. Shorten. A critique of recently proposed buffer-sizing strategies. *SIGCOMM Comput. Commun. Rev.*, 37(1):43–48, 2007.