Synchronized Data Acquisition from Web Services Serving at Disparate Rates

D. R. Plante
Department of Mathematics and Computer Science
Stetson University
421 N. Woodland Boulevard
DeLand, FL 32723, USA
Email: dplante@stetson.edu

Abstract—Service-oriented architectures (SOAs) have provided improved means of serving data in business-to-business (B2B) communications. Usually, however, the flow of data from services is limited by the service provider, and different providers may limit to vastly different rates, leading to different quality of services (QoSs) for each. In some cases, the data from these various providers must be synchronized by the service consumer, for example, when that data comes from different marketplaces (e.g. Amazon or Ebay) and provides such information as prices, item sales ranks, and quantities. The acquisition of that data from each service must be reasonably synchronized if that data is to be reliably compared. This paper addresses the problems associated with this synchronization when QoS differs greatly between services but the quality of the data and the services themselves vary as well.

Index Terms—Web services, quality of services.

I. INTRODUCTION

A service-oriented architecture (SOA) [1] provides a loosely coupled architectural style that facilitates the exchange of data between software agents. In business-to-business (B2B) communications, service providers and service consumers exchange data conforming to specifications determined by the provider. In exchanging this data, the services of different providers may offer vastly different levels of quality of service (QoS). While network lag is one limiter of service, some services are simply not able to provide data at a level that other providers do. It is also the case that some providers purposefully limit or throttle consumer requests at specified rates to prevent abuse of the service and server overload. Both Amazon [2] and eBay [3] provide detailed and specific throttling or call limits on accessing data from their services. Large sellers on their respective marketplaces may receive increased limits commensurate with their increased sales. However, sales on international marketplaces may differ greatly (for example AmazonCA or AmazonUK), again with QoS for these services differing greatly.

As will be explained in Section II, if all data from all services are of equal importance, and if all services are also of equal importance, the rate of the slowest service produces a hard limit to the overall rate that synchronized data may be obtained from all services. Of course, data may be obtained more rapidly from higher QoS services. The problem, however, involves synchronizing each related data unit from each service with that from the other services.

As a concrete example of the problem at hand, assume a particular model camera is selling on eBayUS, eBayUK, AmazonUS, and AmazonUK and the service consumer wishes to obtain the lowest selling price of the camera from any seller on any of those marketplaces. As marketplaces and item prices are often highly dynamic, reliable price comparison requires requesting a data unit (e.g. lowest price of camera for given UPC or model number) from each of the four marketplaces at reasonably synchronized times. Note that the term synchronization here is somewhat relaxed, as the exact time of data acquisition is not required. However, it is important that the data is obtained at reasonably similar times. Reasonably here depends a great deal on the data unit itself, as some item prices on these marketplaces will hardly change at all for days or weeks, while some will change often, possibly in the order of seconds or minutes.

Given the dynamic nature of such marketplaces and QoS limits, it is impossible for any service consumer with large data unit needs to be in relative synchronization with the service provider. For a seller with millions of items for sale in its inventory, the problem can be quite severe. If such a seller wishes to list an item on the marketplace that provides the greatest profit at a given time, knowing other sellers’ prices for items on those marketplaces is critical. Stale price data may lead to reduced profits or worse, to lost sales for not being competitively priced.

In Section II, we briefly provide some background and some related work, though we have found no other comparable work on this problem in the literature. Also in Section II, we describe the problem in more detail. We then describe our approach to a reasonable solution of the problem in Section III. In Section IV we demonstrate the implementation of algorithms developed in this paper. In Section V, we summarize the paper along with some future directions for work on this problem.

II. BACKGROUND AND PROBLEM DESCRIPTION

Online marketplace auctions have been the subject of research for over a decade [4], [5] and continue to be actively studied. Most of this work is aimed at trying to understand underlying consumer behavior and/or price trends, unlike the
present work where we wish to simply have accurate and timely data from multiple services providing data at disparate rates. Also, while previous research has attempted to understand underlying web services, contracts they satisfy, and the QoS guarantees [6], [7], [8], this study focuses on the added constraint that the data obtained from the provider must be reasonably synchronized and obtained from all the services, not only some.

A. Disparate Service Rates and Quality

To illustrate the problem, assume 4 service providers hold 8 instances each of related data, A, B, C, ..., H as shown in Figure 1. For example, data unit A may be the price of a particular model camera, B may be the price of a DVD, etc. Importantly, for all four services, A refers to the same item. While the price may differ, the price is tied to the particular item such that prices for that item may be compared across marketplaces. Assume the QoS of each service permits them to serve data to the provider at a rate of 1, 2, 4, and 8 times (i.e. 1×, 2×, 4×, and 8×) the speed of the slowest service, which in the figure is the first or top service. As the fastest service provides data at 8 times the speed of the slowest, by the time the first is able to serve its second unit of data (i.e. data unit B), the fastest service will serve all 8 data units. The next slowest service will, in the same time frame, serve 4 units, and the next slowest will serve 2 units. One key constraint, however, is that to be useful for comparison, the data must be obtained at reasonably similar times. For example, if the service operating at 2× speed is to first serve data units A and then H and the 1× service serves unit A through H in that order, by the time the 1× service gets to unit H, the corresponding data from service 2× will be stale and possibly inaccurate. Of course, since the 2× service is twice as fast as the 1× service, it may serve the H data again to try and synchronize with the 1× service when that service provides the unit H.

One important point to note here is that no matter how fast the other services are, while they may offer all of the data being provided by the service operating at 1× speed, ultimately, the 1× service is a bottleneck that cannot be overcome when the constraint of synchronization is imposed. Of course, the relaxation of this constraint will help. If, for example, the 2× service serves data units A and B in the time the 1× service serves only A, if the 1× service then serves B to the consumer, the consumer may deem the data unit B from the 2× service to be “recent enough” to use when it receives the related data from the 1× service. By categorizing the data units based on some quality criteria (e.g. the sales rank of the item on the marketplace or the user feedback on that item) and rate of change of that data (e.g. how quickly the price of the item is expected to change), intelligent choices may be made on how often data should be updated.

In this paper, we will not address the issue of how delayed data may be related. Instead, we assume the service consumer requests data from the number of services and then compares it. One constraint that is relaxed, however, is that of service quality. In Figure 1, assume that the 1× service is slow partly because it is a “lower quality” service. For example, it may be that it is a marketplace with fewer sales (e.g. eBayUK having fewer sales per day than AmazonUS). Possibly fewer resources are dedicated to the slower service because it is less active. In any case, it may be that the service consumer deems the quality of the 1× server to be lower than that of the 2× server. Note that the 2×, 4×, and 8× services can all serve data units A and E in the time it takes the 1× service to serve only A. The data units A and E are now synchronized between all but the 1× service. Likewise, the 4× and 8× services both can serve the units A, C, E, and G in the time it takes the 1× service to serve only A and the 2× service to serve only A and E. The circled data units in the Figure 1 indicate these elements. The 8× service could have all 8 data units circled, but the other 4 units would not be synchronized with any other service. (Obtaining that data for analytics purposes would be reasonably, of course.) This may be considered an acceptable solution in some cases, namely to allow services 4× and 8× to keep the 4 data units A, C, E, and G synchronized while the 2× service keeps A and E synchronized with the faster services and the 1× service only keeping the unit A synchronized.

In Figure 1, the squared data units show the same scenario for the next data requests, namely when the 1× service requests element B. We see that in the time it takes the 1× service to obtain two data units, the 4× and 8× services have obtained all 8 data elements in relative synchronization, and the 2× service has obtained half the data units in synchronization with the two faster services. Here it is clear that the orders of speed are important, since having respective speeds of, for example, 1×, 1.7×, 2.3×, and 7.4× would not permit such synchronization. Data units are acquired more rapidly from the faster services, but keeping them aligned at these rates is problematic. However, using other integer powers (e.g. 1×, 3×, 9×, 27×) will also allow for synchronization, especially in cases where the disparate QoS of services are even more dramatic. Therefore, when relaxing the constraint of all services having all data units synchronized, the above method of acquiring data at rates that are integer powers of an integer base is useful.

B. Variable Data Quality

Not all data units served by a service are of equal quality. Using our example of online marketplaces, some items are top sellers, for example a new release DVD of an Oscar
winning movie or a New York Times top selling book. Also, some may even have metrics to indicate their importance on a marketplace (e.g. Amazon Sales Ranks or buyer feedback ratings). As already described, while the slowest service being accessed by a service consumer provides a bottleneck when comparing synchronized data, some improvements to the most restrictive model are possible. By taking into account data quality, another alternative means of acquiring synchronized data is possible.

Assume a single service with the eight units of data from the previous section, A through H. If units A, B, and C are fast-changing and high quality (e.g. hot selling movie DVDs), by cycling through the data in sequence, each unit will only be requested and updated once each in one cycle, the same amount as the other data units. If, however, units G and H are very slow-moving inventory items (e.g. old B-rated movie DVDs) whose prices are fairly unchanging, we may be willing to obtain that data fewer than one time per eight requests so that we may obtain the higher-quality data more often.

In Figure 2, we provide data quality measures to each data unit. From this data, we see that A is considered 4x as important as H. We see that rather than obtain each of the eight data elements in one cycle, we can use this metric to request A four times as often as H. The total number of requests for units A through H is 16. So while A will be requested 4x in the 16 requests, H will only be requested once. However, given the importance of A, this may be a reasonable choice. For the proposed quality measures, a single cycle is now 16 requests rather than the original 8, so the expansion factor for one pass is 16/8 = 2.

![Fig. 2. Data of different quality/importance being served more often.](image)

III. PROPOSED METHODOLOGY

As Figure 3 shows, N service providers serve data units to one service consumer at the request of the consumer. Each provider is considered to have a service quality given by

$$\mathcal{S} = \{s_1, s_2, ..., s_N\}. \quad (1)$$

Service quality here refers to the value placed on the given service and the collective importance of its data, not quality of an individual data unit. The rate at which each service returns data is given by its velocity,

$$\mathcal{V} = \{v_i : i \in \{1, ..., |\mathcal{S}|\}\}. \quad (2)$$

The velocity is a real-valued measure of the number of times faster each service is than the slowest service. By definition, the slowest service will have a velocity of 1.0 while other services will have velocities greater than or equal to 1. As detailed in Section I, since synchronization of data acquired from the different services is key, velocity in the form provided by \(\mathcal{V}\) is not as useful a measure as is the order of the service. The order of the services is defined as

$$\mathcal{O} = \{o_i : i \in \{1, ..., |\mathcal{S}|\}\} \quad (3)$$

where for base \(b\),

$$o_i = \{b^j : j \in \mathbb{N}_0, v_i < b^{j+1}, v_i \geq b^j\} \quad (4)$$

In general, a base of 2 provides the greatest flexibility in optimizing synchronization and speed of services, though higher valued bases may be chosen.

Each service is capable of delivering \(M\) units of data that are to be compared to related data delivered by the other services, where the data provided by service \(j\) is given by

$$\mathcal{D}_j = \{d_{1j}, d_{2j}, ..., d_{Mj}\}. \quad (5)$$

This data may take any form that may be parsed and processed on the receiving end of transmission. For example, data may be flat or comma separated or more structured, such as JSON, or XML. Each unit of data also has a corresponding quality

$$\mathcal{Q}_j = \{q_{1j}, q_{2j}, ..., q_{Mj}\}. \quad (6)$$

We require that each unit \(d_{ij}\) be aligned across all services \(j\) for each unit \(i\). For example, the first unit of data for service \(j\), namely \(d_{1j}\), should correspond to the same data for all services. An example of this may be the price for a given item with a given UPC or ISBN. While the price of that item may differ between different services, the data should be for the identical item (i.e. UPC or ISBN) across services. Likewise, the quality of that unit of data, in this case \(q_{1j}\), should also align with the data for the given service. Using the example of an item with UPC or ISBN, quality is numeric and may be the item’s sales rank on a marketplace or user feedback value for that item. As their values are clearly tied to each other, the data and its quality are combined to produce pairs defined by

$$\mathcal{P} = \{(d_{ij}, q_{ij}) : i \in \{1, ..., |\mathcal{D}|\}, j \in \{1, ..., |\mathcal{S}|\}\} \quad (7)$$

We use the differences in qualities to request data from services in a manner that allows improved performance for higher quality data and services at the expense of lower quality data and services. To do this, we first require that qualities are normalized, namely

$$q_{ij} = \frac{q_{ij}}{\sum_{k=1}^{N} q_{kj}} \quad (8)$$
If all services are deemed to be of comparable quality, the normalized quality of each service is \( \hat{s}_i = 1/|S| \) for \( j \in 1, ..., |S| \). While data quality thus far has referred to data from each different service, for decision making on which data to request more often, an overall quality metric is needed for each data unit collectively across services. This is achieved by weighting normalized data quality by normalized service quality, with

\[
\hat{Q}_i = \frac{\sum_{j=1}^{N} \hat{s}_j}{\sum_{j=1}^{N} s_j}
\]

Clearly for collections of data where \( M \) is large, each value of \( \hat{Q}_i \) will be necessarily quite small. A mapping is desired that will provide a relative multiple of updates per data unit in one complete pass through all data from each service. One way to accomplish this is to map the distribution of \( \hat{Q}_i \) to the interval \([0, 1]\) using a parameterized function and to scale this mapping. Assuming \( \hat{Q}_{\text{min}} = \min \hat{Q} \) and \( \hat{Q}_{\text{max}} = \max \hat{Q} \), we have

\[
f(x) = D[\hat{Q}_{\text{min}}, \hat{Q}_{\text{max}}](x) \rightarrow [0, 1]
\]

\[
c_i = |(C_{\text{max}} - 1)f(\hat{Q}_i)| + 1
\]

\[
C = \sum_{i=1}^{M} c_i.
\]

Here, \( C_{\text{max}} \) is chosen to be the maximum number of times any data item may be requested during a full pass through all data from all services. For a uniformly distributed set \( \hat{Q} \), we note that \( D[\alpha, \beta](x) = U[\alpha, \beta](x) \). The number of data units requested for one complete pass through \( D \) from all services will be expanded by a factor of \( T = C/M \).

One sample mapping is to use a simple linear function,

\[
D[\hat{Q}_{\text{min}}, \hat{Q}_{\text{max}}](x) = \frac{x - \hat{Q}_{\text{min}}}{\hat{Q}_{\text{max}} - \hat{Q}_{\text{min}}}. \tag{15}
\]

Such a mapping may not be desirable, however, if data of higher quality collectively is far preferred over data of much lower quality. In such cases, a mapping such as a logistic function may be preferred, for example

\[
D[\hat{Q}_{\text{min}}, \hat{Q}_{\text{max}}](x) = \frac{1}{1 + e^{-(x - \hat{Q}_{\text{min}})/(\hat{Q}_{\text{max}} - \hat{Q}_{\text{min}})}}. \tag{16}
\]

Since for each data unit \( d_{ij} \) it is required that index \( i \) refers to the same data description (e.g. \( d_{ij} \) is the same book for each service, with only its variable properties, such as price, differing for each service \( j \), we define the request description to be \( r_i \). Therefore, a request for data \( r_i \) will result in data unit \( d_{11} \) being provided by service 1, \( d_{12} \) being provided by service 2, and so on. The request units and their pairings with their respective counts are given by

\[
R = \{ r_i : i \in \{1, ..., |D|\} \}
\]

\[
C = \{ (r_i, c_i) : i \in \{1, ..., |D|\} \}
\]

Algorithm 1: Algorithm to create ordered list of request units.

| Data: set \( C \); number of data units \( M \); max count for a request \( C_{\text{max}} \); total count \( C \) |
| Result: \( \text{requestList} \) ordered list of requests for services |

1. \( \text{requestMultiplicities}[] \leftarrow \text{OrderRequests}(C) \)
2. \( \text{numBins} \leftarrow C_{\text{max}} \)
3. \( \text{binSize} \leftarrow \left\lfloor \frac{C}{\text{numBins}} \right\rfloor \)
4. \( \text{bins} \leftarrow \text{string}[\text{numBins}] \)
5. \( \text{remainderBin} \leftarrow \text{string}[] \)
6. \( \text{requestList} \leftarrow \text{string}[C] \)
7. for \( i \leftarrow 1 \) to \( M \) do
   8. \( \text{request} \leftarrow \text{requestMultiplicities}[i], r \)
   9. \( \text{jump} \leftarrow \text{\{\text{numBins}\}count} \)
   10. \( \text{remainderBin} \leftarrow \text{\{\text{startBinNum} + 1}\}} \)
   11. while \( \text{startBinNum} \leq \text{numBins} \) and \( \text{size(bins[]startBinNum)} \geq \text{binSize} \) do
   12. \( \text{startBinNum} \leftarrow \text{startBinNum} + 1 \)
   13. end
   14. \( \text{binNum} \leftarrow \text{startBinNum} \)
   15. \( \text{lastBin} \leftarrow \text{startBinNum} + (\text{count} - 1) * \text{jump} \)
   16. if \( \text{lastBin} > \text{numBins} \) then
   17. \( \text{jump} = 1; \)
   18. end
   19. for \( j \leftarrow 1 \) to \( \text{count} \) do
   20. \( \text{if binNum} \leq \text{numBins} \) then
   21. \( \text{place copy of request in bins[]binNum} \)
   22. end
   23. \( \text{else} \)
   24. \( \text{place copy of request in remainderBin} \)
   25. end
   26. \( \text{binNum} \leftarrow \text{binNum} + \text{jump} \)
   27. end
   28. end
   29. for \( k \leftarrow 1 \) to \( \text{numBins} \) do
   30. \( \text{foreach bins[]k as request do} \)
   31. \( \text{requestList[]} \leftarrow \text{request} \)
   32. end
   33. end
   34. \( \text{foreach remainderBin as request do} \)
   35. \( \text{requestList[]} \leftarrow \text{request} \)
   36. end
37. return \( \text{requestList} \)

A. Constructing Request List of Multiples

In Section II-B, we gave a simple example of using the quality of data to construct a new list of data unit requests.
In Algorithm 1, we provide one approach to distributing the multiple quantities of data in $C$ based on each element’s respective count, $c_i$. The method described here assumes that given the sizes defined by $l = \lceil \max \mathcal{O} \rceil$ and $n = \lfloor \frac{C}{\max c} \rfloor$, we require that $n \gg l$. Physically, $n$ is the number of request units in each of the $\max c$ bins that data will be placed into and $l$ is the maximum factor of velocity (e.g. 2, 4, 8, etc.) indicating how much faster the fastest service is than the slowest service. Therefore, the required condition will always be met assuming there is a substantial amount of data that is required from the services.

The algorithm works as follows. We first call $\text{OrderRequests}(C)$ so that the data/count pairs in $C$ are in decreasing order based on the value of $c_i$ so that the first pair has the highest count and the last pair has the lowest count. Since multiple pairs will have the same count, the order of the multiples does not matter as long as they all are grouped together. We then create $\text{numBins}$ bins and one overflow bin, $\text{remainderBin}$. Since the number of bins is equal to the maximum multiplicity of request units, we ensure that multiple copies of requests are dispersed across the request list to allow price updates to be separated temporally. For each request-count pair $(r_i, c_i)$, a $\text{jump}$ factor is calculated to allow maximal dispersal of higher-priority requests, indicating how many bins to jump before placing the next copy of the request in the list. An example demonstrating the algorithm is provided in Section IV.

### B. Traversing Request List of Multiples

For Algorithm 2, the $\text{requestList}$ constructed by Algorithm 1 is used to construct each service’s request group. An example of this is illustrated in Figure 1 for a simplified case, where the circled elements form the request group for the services in the first step, and the squared elements are those requested in the second step. This process is stepped along until the slowest service traverses the entire $\text{requestList}$ once. The $\text{jumps}$ being made for each service indicates how many elements of $\text{requestList}$ to step over using the Algorithm 2. Conditionals in the inner loops work to restrict the number of elements being grouped based on speed of service and running beyond the length of $\text{requestList}$.

Part of the request synchronization takes place here at the end of the algorithm, where the requests are made to the services. All requests are made by the consumer, and only once all results are returned by the service providers does the algorithm step along to the next set of requests. Service failure is not addressed and is somewhat application dependent, as business logic must be used to decide how best to handle the problem. It should be noted that one advantage to the approach developed here is that since service velocities are always dropped to match either $2^s$, $3^s$, or other elements of a power series, some flexibility is already built into the model to handle some network lag or delays of these faster services. When the slowest service suffers from some (possibly serious) delays, synchronization of all services becomes problematic in any case as that service already is the bottleneck.

### IV. Example

While in general, the number of data units $M$ in real-world applications will be very large, we illustrate the implementation of the two algorithms from Section III with a small dataset to understand its essence. In Table I, we simplify the structure of the requests using the notation of letter-number, where letter refers to a group of requests having a particular count (e.g. A has a count of 6 copies in the final request list), and number refers to each instance of a unique request (e.g. A4 is the fourth unique request that has a count of 6.) In Figure 4, we show how the requests from Table I are loaded into the final request list by binning that data. As the maximum count

<table>
<thead>
<tr>
<th>$r$</th>
<th>$c$</th>
<th>Number of Unique Requests</th>
<th>Number of Copies In Request List</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 → A10</td>
<td>6</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>B1 → B5</td>
<td>4</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>C1 → C5</td>
<td>3</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>D1 → D10</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

TABLE I

Request-count pairs $(r, c)$ with $M = \text{30 unique requests}, C = \text{105 total list entries and expansion factor of } T = \text{3.5}$. of any unique request in this example is 6, Algorithm 1 begins by creating 6 bins plus an overflow bin of undetermined sizes.
The algorithm places the multiple instances of a unique request unit into bins by jumping an appropriate number of bins. In the present case, for the counts of 6, 4, 3, and 1, the jumps are \( \lfloor \frac{6}{6} \rfloor = 1, \lfloor \frac{4}{6} \rfloor = 1, \lfloor \frac{3}{6} \rfloor = 2, \lfloor \frac{1}{6} \rfloor = 6 \). Therefore, \( A \) units will jump 1 bin, \( B \) units will jump 1 bin, \( C \) units will jump 2 bins, and \( D \) units will jump 6 bins. As we see in Figure 4, \( A \) units are placed in bins 1 through 6, and \( B \) units are placed in bins 1 through 4. However, \( C \) units are placed 2 bins apart if possible, and once units \( C_1 \) and \( C_2 \) are placed in bins 1 and 3, those bins have \( \lfloor \frac{105}{6} \rfloor = 17 \) elements each and are considered full. Therefore, the algorithm steps to the next available bin, \( B \), and places units \( C_3 \) and \( C_4 \) in bins 2 and 4, which are spaced by 2 bins. At this point, bins 1 through 4 are filled, leaving only bins 5, 6 and the overflow bin available. Since jumping 2 bins from the next available bin, \( B \), would move beyond the last of the 6 bins, the jump is changed to 1, and \( C_5 \) is placed in bin 5 then 6. Now, since the last \( C_5 \) unit cannot be placed in one of the 6 primary bins, it is placed in the overflow bin. The 10 \( D \) units are placed in bins 5, 6, then the overflow bin one at a time as bins fill up. These bins are then sequentially loaded into the array \( \text{requestList} \) and returned for Algorithm 2 to request the data from services. This method of placing units gives maximum priority to the highest-ranked data units, with lower-ranked units being placed less optimally if required, as is the case for \( C_5 \) since its 3 copies are not as well spaced across bins.

![Fig. 4. Bin loading.](image)

With the placement of requests in \( \text{requestList} \), the first 17 elements (i.e. those originating from bin 1) are given in Figure 5. If we assume three services operating at speeds (i.e. orders) of 1, 2, and 8, the respective jumps in Algorithm 2 are 8, 4, and 1 respectively and specified in the figure next to each service. Circled units are those that would be grouped for each service in the first iteration of the loop through the \( M \) unique request elements, while the squared elements show which units are grouped during the ninth step through the loop. An important point to note here is that all units in each group are and will always be unique as long as the order of speed is much smaller than the number of data units.

![Fig. 5. Bin traversal.](image)

V. CONCLUSIONS AND FUTURE WORK

In this paper, the problem of a service consumer acquiring data from a collection of service providers operating at disparate rates in a synchronized manner is addressed. We provide one possible solution to this problem by using service and data quality to modify how the data is requested. In this initial approach, variable network lag and service provider failure is not addressed. In future work, the issue of variable network lag for the various services should be addressed, though as already pointed out, some flexibility exists in the present approach to account for some lag. Should serious service quality degradation or inversion take place, for example if the slowest service actually becomes faster than other services due to lag or failure of the other services, the present approach will perform less optimally. The issue of service failure is a more difficult problem to address as how to handle this is somewhat business-model oriented. However, for certain business decisions such as ignore any service that is down, improvements to the present approach are possible, and such effects will be addressed in future work. Also, the present model assumes that all services have all data units available to provide to the service consumer, though this is often not the case in real service applications and is also the subject of future work. Finally, decoupling of the service provider’s synchronized request structure into separate consumer request modules or services may be preferred, with each module requesting data from the providers separately as fast as each can provide it and storing it for use by a data-synchronizing aggregator.

REFERENCES