the SOA magazine
Service Orientation, the Cloud and Beyond
ISSUE XLII AUGUST 2010

Feature Article

Fluid Services (Part II)

by Ilkay Benian

Published: August 9, 2010 (SOA Magazine Issue XLII: August 2010)

Download PDF

Digg This • De.licio.us • Slashdot • Technorati • StumbleUpon • Google Bookmark

The following is the second part of a two part series on Fluid Services. The first part of this article is located here.

Comparing Different Types of Service Reuse

Today's service architectures lead to services that directly talk to data sources and avoid service reuse. This can be achieved either by replicating business rules into each and every isolated service, or by reusing behavior at component level (Figure 5.)

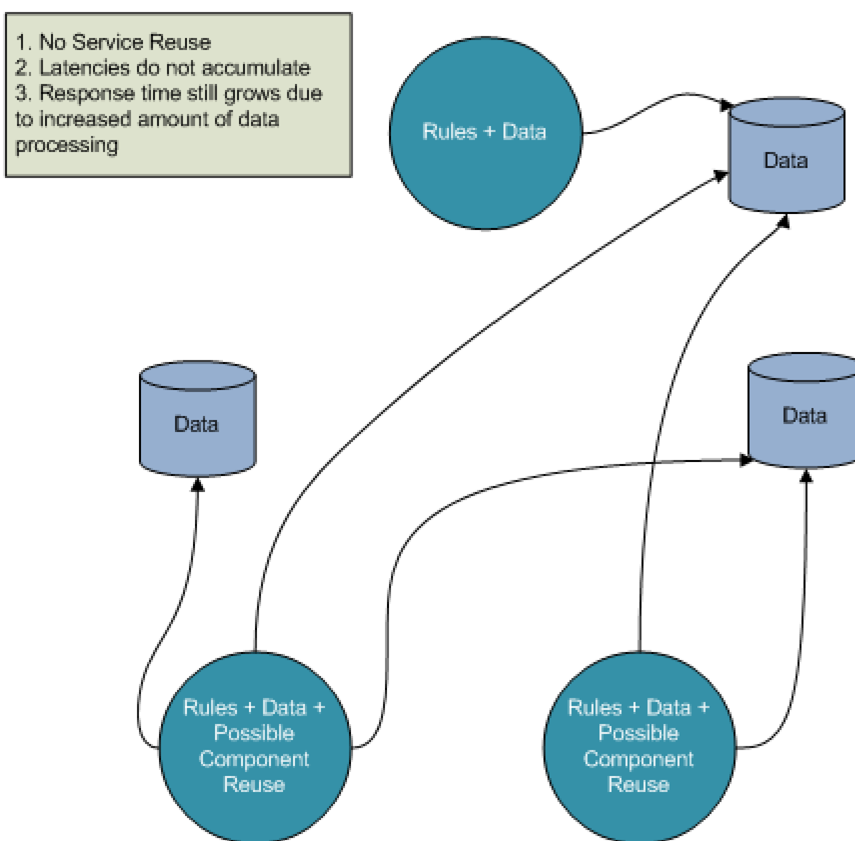


Figure 5

While this design is acceptable in small organizations, it suffers from complexity introduced by deployment and management of reusable components. This is the approach of most component and object oriented methodologies. Data is processed by in memory reusable objects. This design also suffers from increasing response time due to increasing processing.

Figure 6 shows SOA's service reuse approach which basically removes the deployment and version management issues of component oriented and object oriented approaches. However, it has a chronic and terminal illness caused by the accumulating latencies at every service reuse.

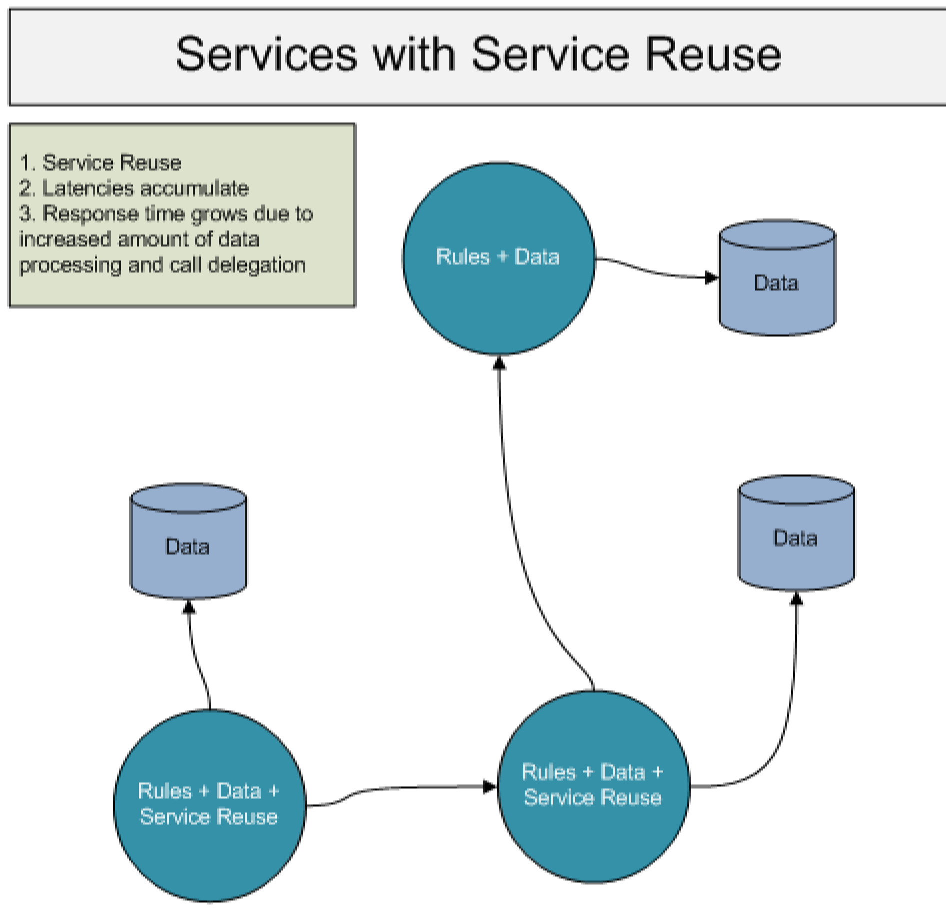


Figure 6

Figure 7 is basically what Fluid Services approach presents as a solution:

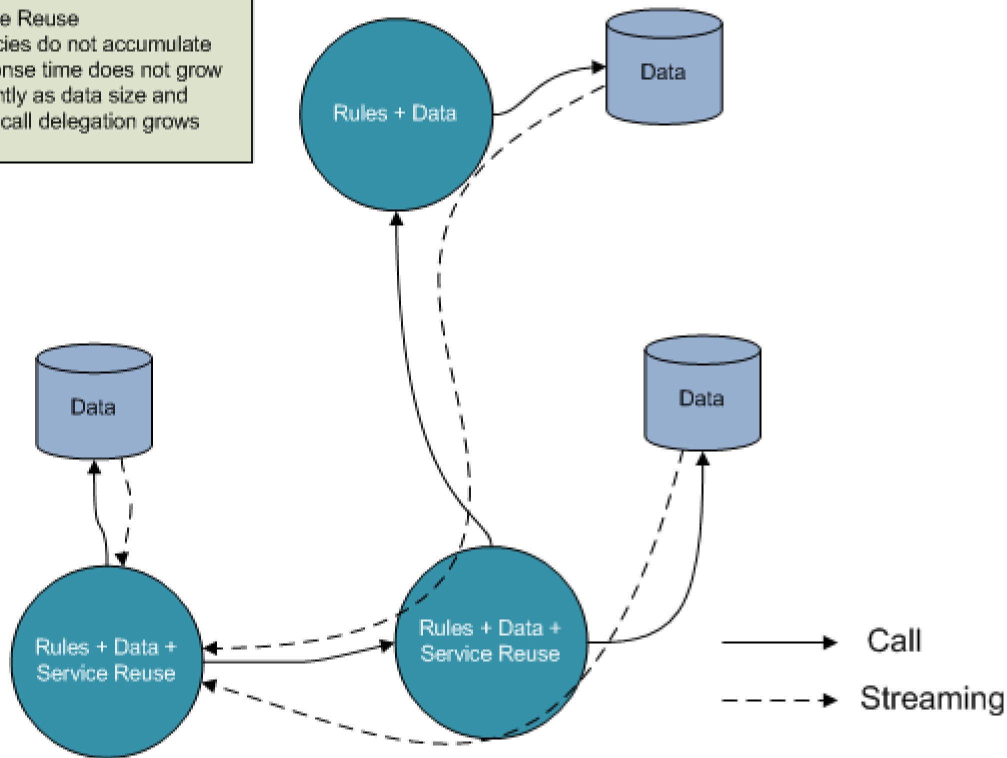


Figure 7

Services are still reused but response times aren't affected significantly, and resources are utilized gracefully. A fluid service call triggers a chain reaction that reaches to the leaf nodes quicker and gets a response quicker. All involved parties keep processing as long as the client opts to continue consuming the response. When the client decides to stop consuming, the services stop again with a chain reaction.

Figure 8 illustrates just one scenario where the Fluid Services would really shine:

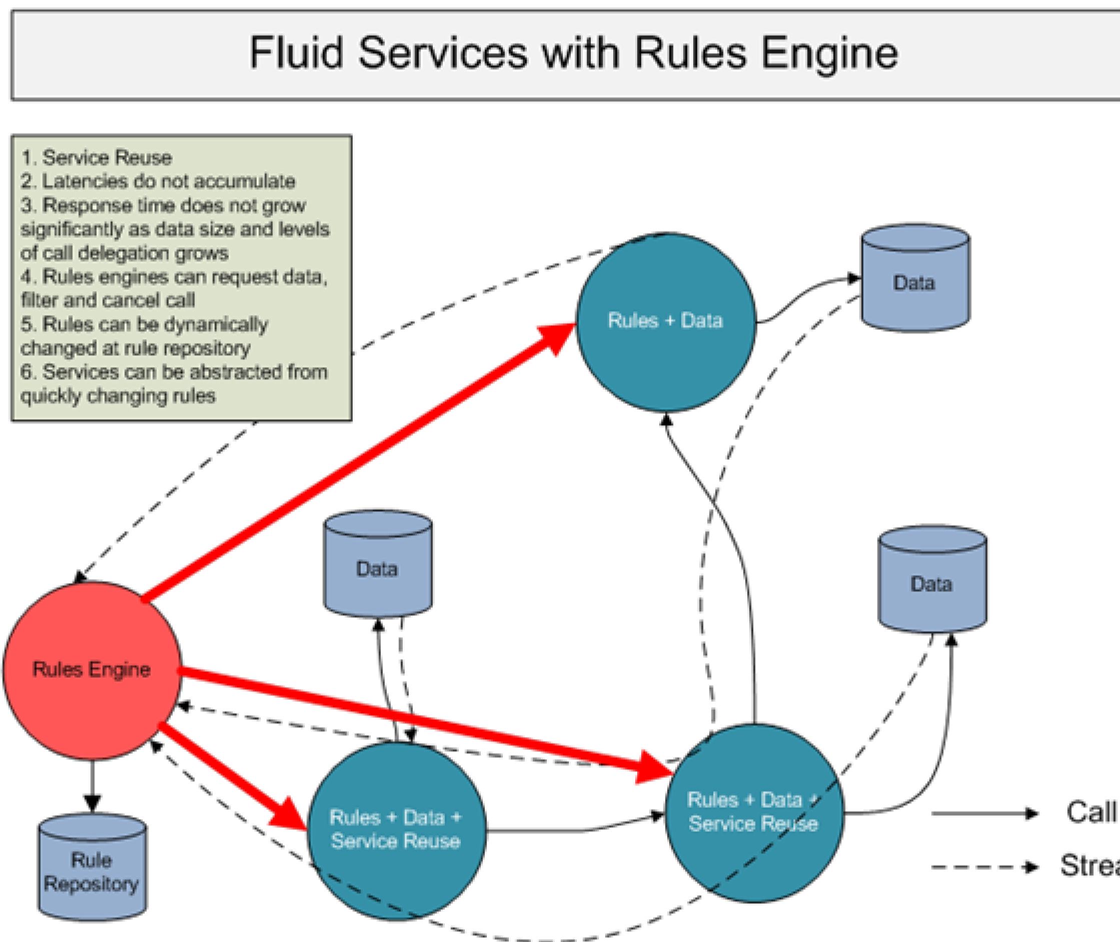


Figure 8

A rules engine is used to get real time data from multiple services and process results. But the back end services may not have been designed for the kinds of rules that are currently being executed by the rules engine. According to the principle of separation of concerns, this is exactly how it's supposed to be. The back end service should have no knowledge of current rule repository, and the rule engine should have no knowledge (other than the contract) of what the internal details of the existing services are. The rules can be stored in a central rule repository and are flexibly and much more quickly changed than the services themselves. Since rules engine can trigger transactions and start consuming the responses quickly and until it meets a certain criterion, it is much more feasible to actually make this scenario work and scale well under growing amounts of load. The same argument can well be made for an orchestration service, for it is almost the same idea, except the rules may not be that flexible.

Services as a Processing Pipeline

The fluid services concept can also be analyzed in terms of the 'pipeline pattern'. When multiple service calls are chained and each service is able to work on partial data during the call, the operations performed at each service (pipeline stage) overlap. A physical analogy is a 'serial assembly line'. Each stage in an assembly line is designed for a specific purpose but is always kept busy by feeding one stage's output into the next stage. This makes sure the assembly throughput is maxed out because no intermediary stage is kept idle until the processing of a single service call is complete.

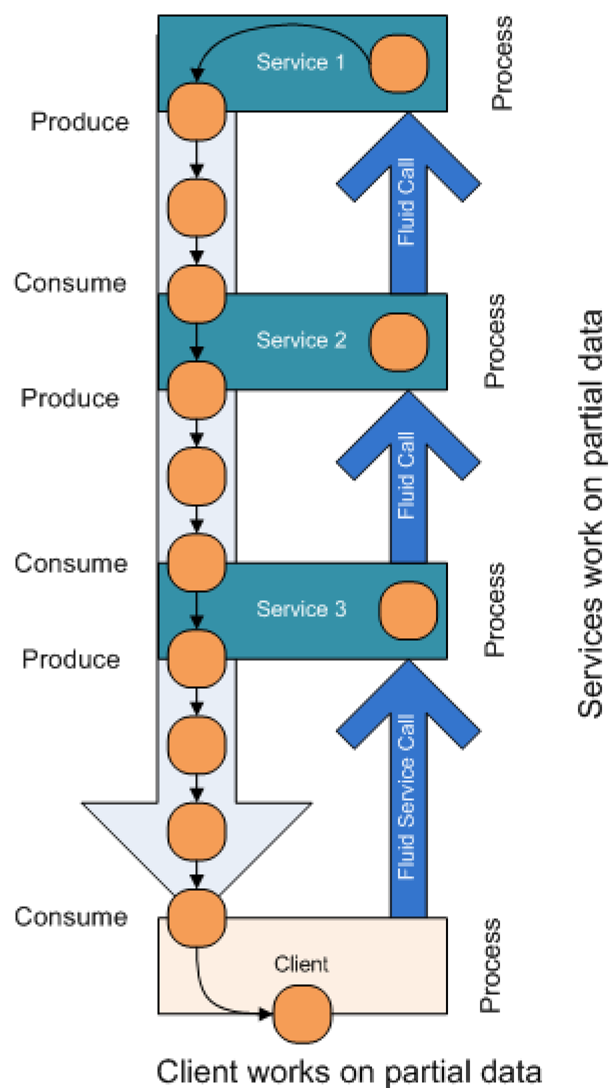
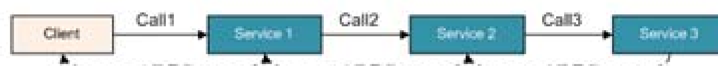


Figure 9

In Figure 9, it takes only 3 stages to complete a single product and we don't even have to wait for the whole production to finish. So the latency is also pretty low. If a serial assembly line were to be designed the way we design web services today, it would have to work on a large batch of items at each stage increasing the stage response time, the overall completion time would stretch out dramatically, and throughput would be reduced. The system could still be relying on the arrival of new batches to keep the system busy. But if there is not enough number of batches to feed the stages, the assembly line would stand idle for long intervals of time. Even if there is enough batches that arrive, that would still not change the total time for a single batch to complete.

Similarly a service can be thought as an assembly stage that is specialized to perform a certain type of task. If we can keep all the stages (services) busy as soon as possible even for a single call, we will be able to make use of the parallelism between them to increase the throughput and reduce the total response time just like in an assembly line.

A chained service call scenario



A simplified view of overlapping that includes processing and transfer buffer latency

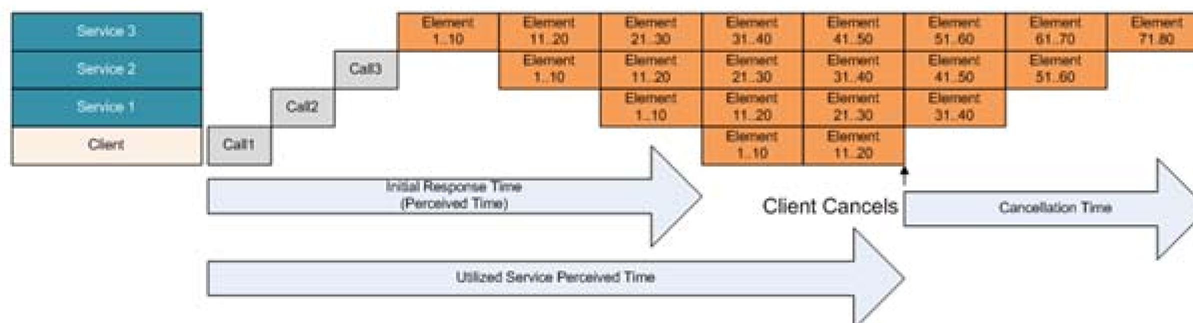


Figure 10

Figure 10 illustrates a simplified view of the overlapped processing steps in which a client triggers a chain of 3 levels of service calls. Despite the call depth, the client is able to start consuming results pretty quickly. Even if the number of elements that the services actually return is large, the perceived response time is still low. This makes such service architectures immune to data size growth problem. Most operations do not really require consumption of all the data returned from services. Rather the data is used by a client selectively, perhaps just to display a limited amount of data, or to find data that matches given criteria.

Even if we pull the entire dataset, we are still better off by utilizing overlapping the produce/consume time because the total time is not multiplied by the number of stages (services). Today's web services on the other hand, has to buffer the data at each service and return the full dataset which means the total time is multiplied by the number of services assuming each service has the same processing time.

Scalability Characteristics of Fluid Services

Fluid services thus make sure services have well scalability characteristics in the following dimensions:

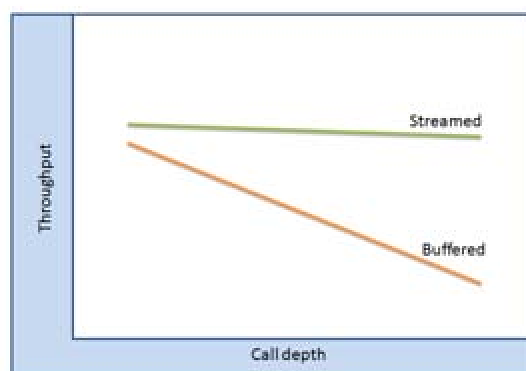


Figure 11: Throughput is kept constant as the call depth grows

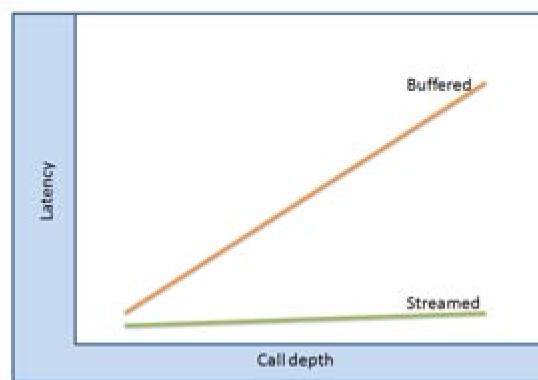


Figure 12: Latency is kept constant as the call depth grows

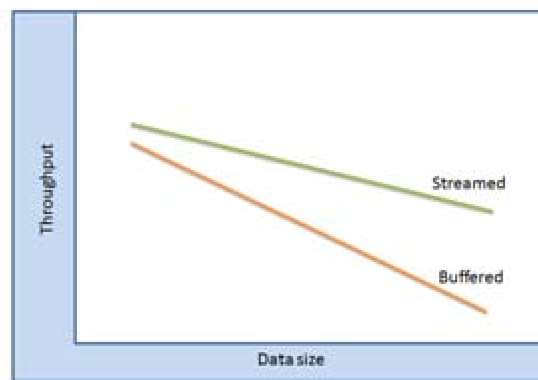


Figure 13: Throughput decreases linearly as the data size grows, but it has a lower slope

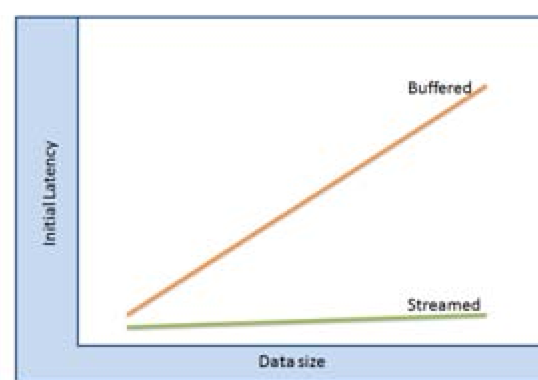


Figure 14: Initial latency is kept constant as the data size grows

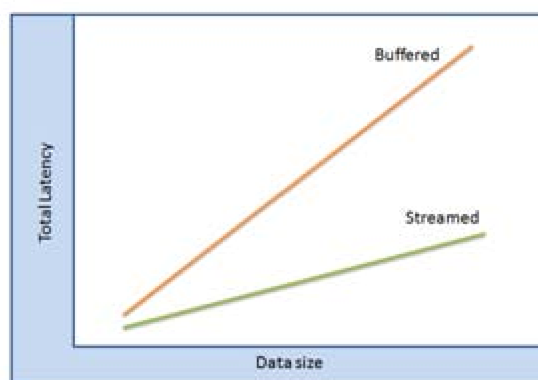


Figure 15: Total latency increases linearly as the data size grows, but it has a lower slope

Buffered services incur latency cost at every stage (service) whereas the streamed services incur latency cost only once because almost all processing time is overlapped.

These are not actual test results but only projections plotted based on a qualitative analysis. The real results may be affected many properties of a system including channel and protocol used, network topology, development platform, operating system and others. The purpose of this qualitative analysis is just to give some sense of what we can expect to gain by adopting the proposed design style.

Some Challenges Facing Fluid Services

Here are some of the important problems that must be addressed before adopting the ‘Fluid Services’ style design for a wide range of future services:

Streaming

The streaming capability is an absolute requirement for fluid services to be of any use. If the platform does not support such capability, any workarounds that may be devised may end up creating more problems than solutions it provides. Streaming or a similar feature is supported by HTTP, TCP, Named Pipes, ADO.NET Async Command Execution. On the other hand, queue based message oriented systems like IBM WebSphere MQ, JMS or MSMQ is less likely to directly support such overlapped processing capacity. A message splitter pattern may provide some workaround for this shortcoming but real life performance characteristics must be analyzed to say something conclusive.

Overlapped IO and processing

Overlapping service processing with serialization and data transfer is the most important first step to designing a fluid service platform. But this is only for service response which is half of the problem. The same design must be implemented by the clients of such services. Namely, they have to overlap client’s processing with service response reception and deserialization time. The serialization and deserialization must also be designed so that it is aware of the multiplicity of data elements in the request or reply. A fluid service should return almost immediately from the service method but keep processing and pushing data into the reply channel as the results are computed or received from another source. A fluid service client should execute consumption code as it receives results from the reply channel. Although this design sounds much more complex than usual serial type of service programming, language syntactic sugar (like iterators, yield keyword, continuation pattern) can greatly reduce the complexity.

Call Cancellation

Although not an absolute requirement, call cancellation is one of the neatest features that fluid services can potentially benefit. Depending on the channel type and protocol call cancellation may or may not be possible. A fluid services design strategy with robust support for call cancellation will definitely pay for itself enabling better scalability, composition, distributed processing, rules engine processing, orchestration as well as better end-client response characteristics.

Holistic Aggregators, Holistic Rules

The fluid services require that each node in the system is capable of processing partial data. This is usually something like a domain entity or other complex type that is returned as a sequence of elements in a streamed fashion. However, there are certain types of holistic computation that requires availability of the entire dataset to work on. Some examples are:

Sorting, filtering, vertical calculations like averaging, summing etc, per element processing, processing that has persistent side effects and should not be interrupted in the middle.

Some solutions may be developed to address the above types of operations to be performed while still keeping fluid services style of design. Since these types of operations require some or all of the data to be buffered before executing, it might be a good practice to just defer the buffering to the client so that intermediary and back-end services all work in parallel using streamed communications.

Sorting is a particularly challenging problem: If there are multiple back end services that are called and aggregated, a resequencer pattern may be employed to buffer some of the data and send only after making sure correct order is achieved. If each back end service were to return its own data in the same sort order, then the resequencer wouldn't have to wait for the whole data set to complete.

If it is not possible to solve the problem by partial buffering of the data, and a full buffering is inevitable then the architecture could be designed so that the buffering occurs only once (or a minimum constant number of times) in the service call chain. This type of processing would be best handled as close to the end-client as possible, and the back end service call chain could just work with unordered data. This will still ensure that the full buffering cost is incurred once (or at least a minimal constant number of times) and adding new services will keep response time constant.

Vertical operations like average, sum, count, max and min etc also requires special handling. But these operations do not really require buffering because of their commutative nature. All such commutative operations could be processed as they pass through a processing pipeline stage. The only requirement is probably that all elements are taken into calculation. Therefore, this type of computation will eliminate the possibility of call cancellation but it is still possible to keep response time constant against call-chaining.

Fortunately, most service calls do not require such complex processing and will be able to process data as it is available and will also allow cancellation. But still, a real world fluid services implementation will need to address this issue intelligently taking into account all other technical considerations of the platform(s) that it targets.

Conclusion

By solving the call latency, throughput, and call cancellation issues the reuse/composability characteristics of services will be greatly improved, and it will be much more feasible to actually start reusing services to create other services which will in turn reduce development and maintenance costs significantly. Evolving service-oriented systems will be able to catch up with growing workload, data size and functionality without impairing the performance which will greatly increase the value of existing services.

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

Copyright © 2006-2010 SOA Systems Inc.