Service Oriented Database Architecture: App Server-Lite?

David Campbell

June 2005

Technical Report
MSR-TR-2005-129

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Service Oriented Database Architecture: App Server-Lite?

David Campbell
Microsoft Corporation
1 Microsoft Way / 36821
Redmond, WA, USA 98052

davidc@microsoft.com

ABSTRACT

As the capabilities and service levels of enterprise database systems have evolved, they have collided with incumbent technologies such as TP-Monitors or Message Oriented Middleware (MOM). We believe this trend will continue and have architected the upcoming release of SQL Server to advance this technology trend. This paper describes the Service Oriented Database Architecture (SODA) developed for the Microsoft SQL Server DBMS. First, it motivates the need for building Service Oriented Architecture (SOA) features directly into a database engine. Second, it describes a set of features in SQL Server that have been designed for SOA use. Finally, it concludes with some thoughts on how SODA can enable multiple service deployment topologies.

1. INTRODUCTION

Three to five years ago, the SQL Server architects saw large scale loosely coupled system architectures emerging as a natural pattern; particularly in Internet scale transaction processing systems. We observed a natural evolutionary pattern as many of these large systems ran into scaling and availability issues with their first and second generation architectures that were often built on tightly coupled, large scale platforms with monolithic, and sometimes clustered, databases at their core. Many of these successful systems are now based upon a loosely coupled, service centric, architecture. With loose coupling and service based partitioning, large systems exhibit more resilience to failure and resources may be added incrementally throughout the architecture as needs change. This presents a new set of challenges to enterprise database systems supporting these scenarios.

Observing this shift, the SQL Server team responded by adding a number of specific features to SQL Server to address the needs of Service Oriented Architectures. Collectively, these features implement what we refer to as a Service Oriented Database Architecture. We believe many of the features associated with SODA will ultimately become as commonplace in database systems as network communications, thread pooling, and stored procedure logic have become.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2005, June 14–16, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00.

In this talk, we describe the architecture behind four of the SOA features available in the upcoming release of SQL Server:

- *SQLCLR:* Embedding the Common Language Runtime (CLR) in the core database engine. Deep integration is the key to success; a hosting layer between the SQL engine and the CLR allows resources, such as memory and threads, to be coordinated between the runtime and database engine. System or user developed assemblies are stored and loaded directly from the database rather than the file system reducing administrative complexity. A modern development environment allows simple deployment and debugging of user developed code.
- *Database Change Notifications:* A scalable notification mechanism that allows data dependent caches to receive notifications when underlying database changes invalidate their cached data. Complex queries can be represented so that a web page rendering a product catalog on an e-commerce site could be notified when inventory status changes rather than polling the database to detect changes to the catalog.
- *Native Web Service Access:* SQL Server is integrated with a new kernel mode HTTP driver in Windows 2003 to allow SQL Server to directly register a portion of the URL namespace for its use without requiring Internet Information Services (IIS) as an intermediary. Administrators can publish any stored procedure as a “WebMethod” allowing a SOAP compliant client to consume services directly from the database engine.
- *Service Broker:* The SQL Service Broker represents a completely new class of transactional middleware. Rather than being message centric from the programmer’s perspective, the Service Broker is service centric. We will show how this distinction makes it easier to develop scalable services and allows SQL Server to employ a number of transaction processing techniques to support high scale services.

2. SOA and Databases

Service Oriented Architectures are characterized by loose coupling across service implementations using prescribed service contracts. The service contracts can either be *operation-centric* e.g. publishing a set of operations and the data and operands required for those operations; or *document-centric* in which the operation and operands are encoded in a single “document” that is submitted to the service. In either case, any program that can

construct a properly formatted service request can interact with the service without knowledge of how a given service is implemented. The services themselves are free to process the service request with whatever tools are appropriate. In fact, due to this loose coupling, service implementations can be scaled, evolved and changed dramatically, as long as they continue to implement the specified service contract

Service requests and responses are typically encoded as "messages" and the message format of choice these days is XML. It is important to note that messages and data flowing between services are generally not stored in the same form as data inside a service. This distinction is described by Helland [1].

Thus, a database that directly supports SOA must have the following properties:

- It must provide proper endpoint support to act as a service host – this may imply a TCP Socket, HTTP GET/PUT, SOAP endpoint, or other message processing endpoints.
- It must be able to directly process and transform service requests into a form suitable for processing. In today's world, this implies the ability to process XML with ease.
- It must provide an efficient and easily programmed service logic host. This includes many features associated with modern application servers including pooling, activation, and ability to scale out logic processing.

With these properties in mind, the next release of Microsoft SQL Server was build to implement a Service Oriented Database Architecture (SODA).

2.1 Why SOA in the database?

There are a number of reasons for implementing SOA features directly in the database engine – several of them are presented here:

- *Messages are data too:* Service requests and responses are very interesting data. They may be durably queued in the database. In this form, one may want to query, back up, restore and generally manage the data. This argument has been made by Gray [2]. Furthermore, the raw service request and response data may be stored and archived to support subsequent auditing or business analysis.
- *Scale Up/Scale Down:* Services may be instantiated at a wide variety of scales. The ability to consolidate multiple tiers of service logic into a single database process allows for scaling services down without administrative complexity. SQL Server attempts to make the choice of whether to deploy service logic inside the database tier or separate middle-tier a deployment time, rather than a design time decision.
- *Grid Computing:* There are several ways to scale out data centric computing: by scaling out the database itself; generally by clustering options offered by the database vendor, or by distributing the processing via SOA. When scaling out the database, the resultant database cluster is more tightly coupled than the SOA alternative. When we looked at the large scale TP applications, several of them had moved from use of a database cluster to factoring the application via SOA and application level partitioning. Support for SOA directly in the database reduces the number of component processes necessary to build out a true grid solution.

3. SODA Features

This section describes a number of the SQL Server features designed to support the SODA.

3.1 SQLCLR

Service requests are processed within the bounds of a service implementation. For a database system itself to be a credible SOA engine, it must host service logic in an efficient fashion. The SQLCLR feature integrates the Microsoft Common Language Runtime (CLR) inside the SQL Server runtime process. The CLR is a modern runtime environment that supports multiple programming languages such as C#, Managed C++, Visual Basic, COBOL and others. As a managed environment, the CLR supports garbage collected memory management, application logic pooling, and system resource management. The core SQL Server runtime environment supports execution of its native SQL dialect Transact-SQL as well as hosting user written native code through Extended Stored Procedures. Since both of these environments execute in a common operating system process, managing the interactions between them is essential. To this end, we have integrated the two runtime environments by having the CLR execute within the SQCLR Hosting interface as shown in Figure 1. This hosting interface allows the SQL Server process to work with the CLR runtime to control global process resources such as memory, threads, and concurrency primitives. For more information on SQLCLR integration, see Blakeley, et. al. [3].

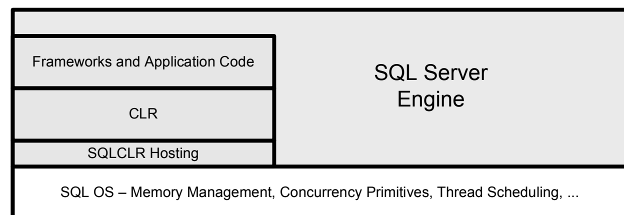

Figure 1: SQL CLR Hosting Layer

The SQLCLR feature allows development and deployment of new types, aggregates, triggers, as well as functions and procedures. Of particular interest to SOA are functions and procedures which interact with data in the database. SQLCLR includes a managed client data access library which can run either within the database process or in a standard client configuration in a separate process with some minor differences. This programming model symmetry changes the decision on whether to run the service logic within the database engine or in another process (or machine) to a deployment, rather than a design time decision. When run inside the database engine, latencies and transition costs to acquire data are reduced significantly but the service logic itself may compete with the computing resources available for general database use. When deployed outside the database engine, say on another machine, the service logic can be moved outboard to scale out allowing the database engine to focus on data and message processing for the SOA.

3.2 Database Change Notifications

Many significant database applications cache information from the database server to avoid accessing the database on every request. These caches appear in mid-tier data caches, dynamic web content presentation caches as well as other application specific caches. While the caches offload the database server, they must typically balance consistency and performance as the general refresh policy is often based upon periodic refresh from the back end. Refreshing the cache too frequently will needlessly burden the back end database. Refreshing infrequently can lead to application errors due to data inconsistency between the cache and the "truth" stored in the database.

Database change notifications (DCN) allow a cache client to subscribe to changes in a query result set and receive notification when any underlying change in the database would alter the results of the query the cache has subscribed to. With DCN, the client cache can avoid periodic refresh and can react quickly to changes that invalidate the cache.

The general workflow for a cache using DCN is as follows:

1. A SQL query is submitted to the database server with a special annotation indicating the client is requesting a change notification.
2. A *subscription request* is created and registered for the query. The query is adorned with information to support DCN and subsequently optimized and executed.
3. Query results are returned to the client application (cache).
4. Any underlying change (DML or DDL) which would alter the results of the query cause the database engine to enqueue a *change notification* to the subscribing application[1].
5. The change notification is processed and sent to the subscribing application.

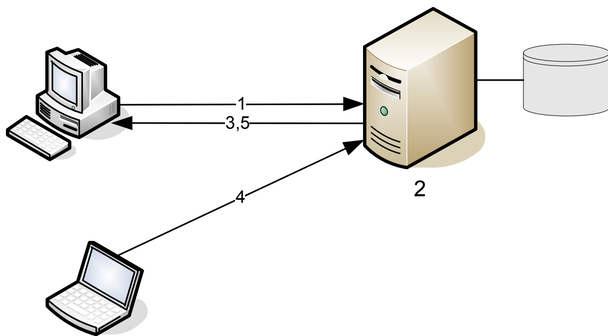

Figure 2: DCN topology and flow

The DCN feature is available to any data access client and has been transparently integrated into the caching mechanism supported by ASP.Net.

[1] The mechanisms used by the database engine for maintaining "indexed views" are used by DCN to determine if subscriptions must be notified in the face of underlying data updates.

3.3 Native Web Service Support

As mentioned previously, the SODA requires that we support appropriate endpoints for accepting and processing service requests. SQL Server now supports a formal endpoint abstraction that can be provisioned to support a variety of connections into the database server including: the native Tabular Data Stream (TDS) protocol; Database Mirroring; SOAP; and Service Broker. In this section we describe SOAP endpoints and the native web service implementation.

Windows 2003 includes a new kernel mode HTTP "listener" as shown in Figure 3. Applications, such as SQL Server, can register with the HTTP listener and reserve portions of the URL namespace for their use (1). Once registered, HTTP requests destined for the registered namespaces are dispatched to the appropriate service application without using Internet Information Services (IIS) as an intermediary (2). This direct dispatch model is highly efficient and provides kernel based process isolation between the service processes owning different portions of the namespace.

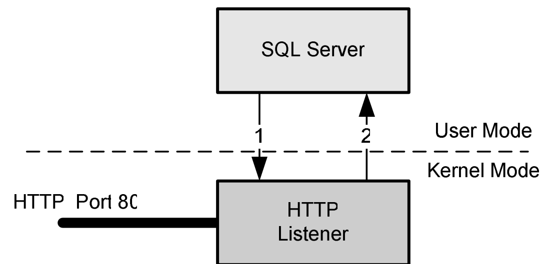

Figure 3: Kernel Mode Listener

Once configured with the HTTP listener, database administrators can provision SOAP endpoints and then bind web service methods (WebMethods) to the endpoint. WebMethods registered through the endpoint can be configured to also generate Web Services Description Language (WSDL) for each exposed WebMethod, or a custom WSDL file may be associated with each WebMethod.

SQL Server, like most other enterprise databases, supports a stored procedure language that allows users to define procedures. SQLCLR extends this to include procedures written in C#, VB.Net and Managed C++. SQL Server's TDS protocol allows these procedures to be invoked by other Transact-SQL code via the EXECUTE statement or explicitly invoked by a protocol level remote procedure call (RPC) that performs efficient parameter binding and execution. This RPC mechanism however, requires a TDS client be installed on the remote system and requires a connected TDS session to invoke and process the RPC.

The combination of the kernel mode HTTP listener and native web services, particularly with automatic WSDL generation provide a standards based RPC mechanism with modern authentication mechanisms to allow any SOAP compliant client to invoke WebMethods exposed from SQL Server.

3.4 Service Broker

A number of database systems have integrated transactional message queuing support in the database engine. This approach has a number of advantages as described by Gray [2]. Rather than

doing message queuing integration with a message broker, SQL Server implemented a service broker. From a programmer's perspective, applications are service centric rather than message or queue centric. Reliable communication dialogs are established between services which may be local or remote. From the system's perspective, service centricity allows the SQL Server engine to handle complex concurrency control of message rendezvous from multiple concurrent service programs and allows a number of transaction processing optimizations.

3.4.1 Service Broker Defined

The service broker defines a number of new database objects including:

- Services
- Queues
- Service Contracts
- Messages

Service broker also extends the Transact-SQL language with new verbs such as:

- BEGIN DIALOG / END DIALOG – for creating a reliable communication channel between services
- SEND – to send messages on a dialog
- RECEIVE – to perform a *(potentially blocking)* read from a dialog.

Service broker also contains an internal activation model that allows binding of service programming logic, expressed in any CLR supported language to services. While service broker does not currently supply an external activator for running service code out of process, it does provide hooks to implement and control an external activator.

3.4.2 Issues with traditional queued message processing.

In traditional queued messaging applications that support message sequences beyond simple request/response patterns, the application programmer must handle correlation and concurrency control between messages destined for a queue. This problem is particularly acute when there are multiple service programs consuming messages from the queue. Assume a service S_1 consumes a sequence of messages $\{M_1, M_2, M_3\}$. These messages are processed by two service programs P_1 and P_2. Further assume that each message must retrieve state from the database when processing each message. With traditional queued processing, as shown in Figure 4, each service program may consume a message that arrives at the queue independently – e.g. M_1 may be consumed and processed by P_1 and while P_1 is active, message M_2 may arrive on the queue and be processed by P_2. Since the messages are correlated, it's quite likely that M_1 should be processed fully before *any* service program consumes M_2. If the processing of M_1 and M_2 requires updating state in the database, concurrent processing by separate service programs will result in blocking or even deadlocking on the shared state.

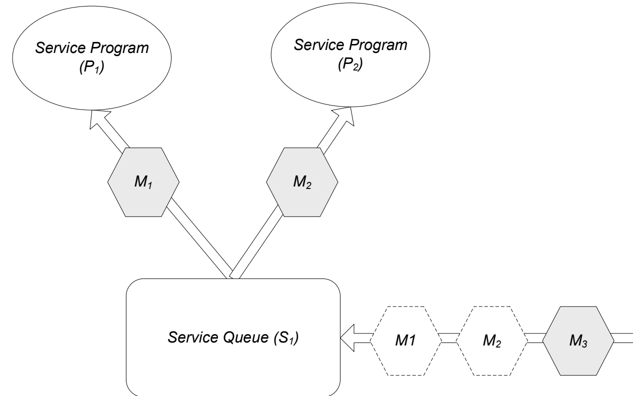

Figure 4: Queued Message Handling

Service broker introduces the concept of *conversation group locking* to address this concurrency issue. Furthermore, in traditional message processing, if state from the database is required to process a message, the service program must draw a message from the queue, interpret the message, (perhaps to retrieve a customer ID or order number), and then form the request to retrieve the state from database. Conversation groups and formalization of services allow service broker to address both of these issues as shown below.

3.4.3 Concurrency and state management with Service Broker

Messages between service broker services are carried upon *dialogs.* Dialogs provide full-duplex, reliable, durable, and in-order message transmission. A service may start multiple dialogs to process a single long running request. For example, an order processing service may receive an order request on a dialog and subsequently create dialogs to an inventory and credit-check service as shown in Figure 5.

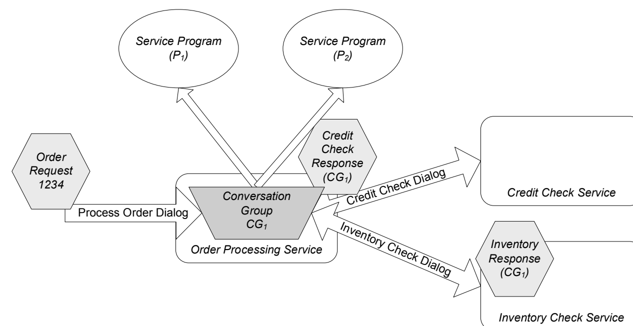

Figure 5: Order Processing Service

All of these dialogs are related in a single conversation group CG_1. Once the service requests have been sent to the inventory and credit-check services, the conversation group state can be committed durably to the database while the inventory and credit-check services are processing the requests. At this point the conversation group is in a pending state awaiting a message to activate the conversation group. This message may come as a response from either the inventory or credit-check service. In fact, responses from both services may arrive somewhat

simultaneously[2]. Once a conversation group has work to be performed it becomes active. With this as background, one should note that service programs wait for conversation groups to become active rather than for messages to arrive. An active conversation group is processed within a transaction that may only be processed by a single service program. In the example we're using, if the response message from the inventory service arrived while a service program was processing the credit-check response, it would not be dispatched to any other service program but rather it would wait until the conversation group was committed and reactivated by any other service program. If both the credit-check and inventory responses arrived before any service program had consumed the active conversation group, the service program would be presented with **both** messages – the messages would be batched and associated with the active conversation group. Furthermore, if the inventory check response arrived while the service program was processing the active conversation group CG_1, the message would be queued to the conversation group and the service program could check for additional work (to process the inventory response) prior to completing its processing of CG_1. This batching leads to very favorable scaling characteristics as work on behalf of a conversation group can accumulate in response to processing latency by service programs. Thus, the service broker quite naturally trades latency for throughput under load much like standard group commit protocols do for transaction logs.

The formal service and conversation group abstractions of the service broker enable optimization of state retrieval from the database. Each conversation group is associated with a globally unique ID (GUID). A state retrieval query may be bound with each conversation group such that each service program that processes an active conversation group not only has the opportunity to receive one or more messages destined for that conversation group but also can receive the results of the state retrieval query **without** having to interpret the messages and then make a separate request to the database engine.

4. Computing topologies enabled via SODA

SODA, as implemented in SQL Server, brings together:

- The query power and ACID transaction properties of enterprise scale databases.
- Native Web Services to connect many web service clients.
- Transaction Services to implement highly reliable, durable, and scalable services.
- A sophisticated data change notification (DCN) feature to build efficient and highly coherent caches.

All of these features, and even the associated service program itself, can be hosted within the context of a single SQL Server process. Since SQL Server scales from laptop to enterprise class servers, this opens up a wide range of computing topologies for software developers. For example, a vendor could choose to implement a solution in a way that could *scale down* to a single process deployment for desktop or notebook configurations and then *scale up* to a standard multi-tier configuration using SQLCLR to run the business logic in either a data-tier or mid-tier configuration as shown in Figure 6. What is novel about using SQLCLR to implement middle-tier data logic is that the deployment configuration becomes a deployment time decision rather than a design time decision[3].

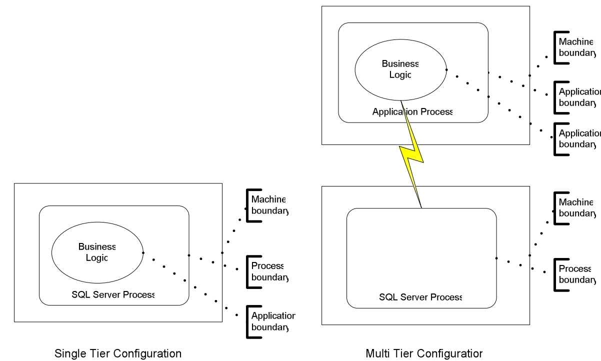

Figure 6: Deployment Configuration Choice

Information workers and field specialists could have services running on their laptop in an occasionally connected model that connect to proxy services which themselves connect to back end services as shown in Figure 7. In this configuration the proxy server acts as a dialog relay between the service client running on the laptop and back end servicing nodes. In many cases, there may be an asymmetric connection topology between the service client and back end processing due to firewalls or other security mechanisms. In this case, the proxy service waits for the client to connect to the proxy server and retrieve any queued messages destined for the service client.

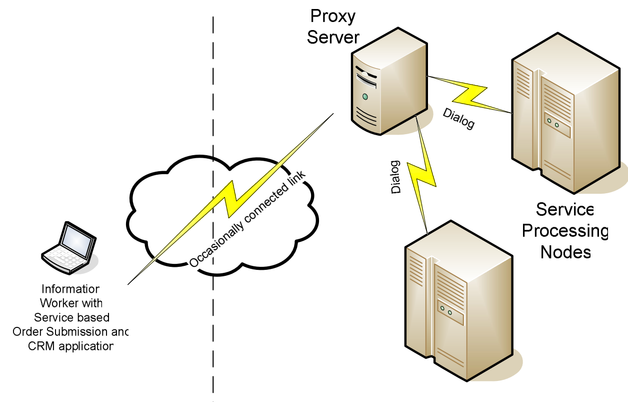

Figure 7: Occasionally Connected Service Topology

The SODA and, in particular, Native Web Services combined with SQLCLR and Service Broker allows us to naturally scale a SOA solution without changing the service contract. This is demonstrated in the following sequence.

Step 1: Service in a process

---

[2] Service broker also allows applications to associate timeouts with each dialog. Timeouts, of course, arrive in messages; so a conversation group may become active as a result of a dialog timing out.

[3] There are some minor differences between the in-process and client data providers but there is a common subset that will allow building application logic that can be hosted either in the mid-tier or data-tier.

In the first deployment scenario, we use native web services, SQLCLR and Service Broker all hosted within the SQL Server process. This is shown in Figure 8.

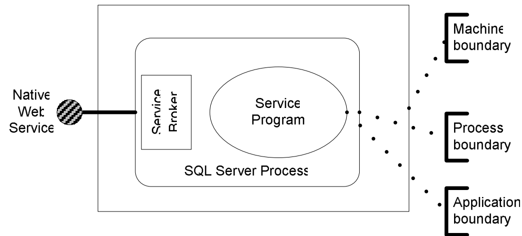

Figure 8: Service in a process

Step 2: Scaling out the service programs

Running the service programs in the database engine significantly reduces the latency between the application logic and the data. However, it consumes processing resources that could otherwise be used for database work. The service broker, coupled with the CLR, allows us to deploy service logic either within the database or run externally on the same machine as the database engine or on another machine altogether. This allows us to scale out service logic and have the database engine focus on service request and data request processing.

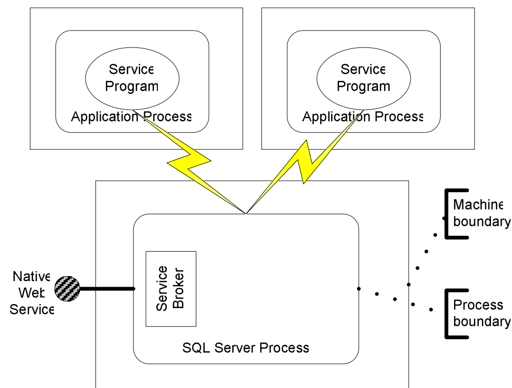

Figure 9: Scaling out service programs

This is demonstrated in Figure 9.

Step 3: Scaling out services

The service broker architecture will allow the introduction of sophisticated service routers which can act as intermediaries on dialog creation to initiate dialogs across a tier of scaled services. Routing policy could include knowledge of data partitioned across tiers. This is shown in Figure 10.

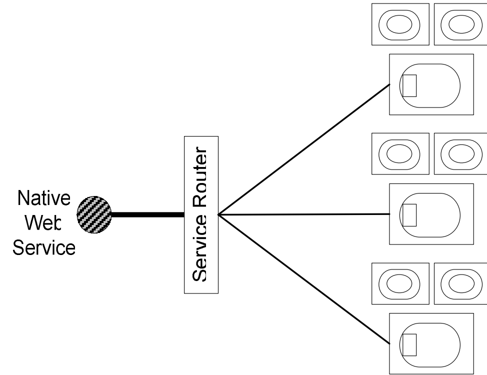

Figure 10: Scaled out services

5. Summary

This paper presented a number of SQL Server features directed at facilitating SOA development. First, we briefly introduced each of these features that allow SQL Server to directly process service requests and form the basis of a core Service Oriented Database Architecture (SODA). Next we described each feature and its contribution to SODA. Finally, we presented several interesting architecture topologies enabled by SODA. While each of these features may be used independently; they are designed to work in a synergistic fashion to build independent and autonomous service processing "bricks" that can be used to construct and compose a number of very interesting application topologies.

Just as TP-Lite emerged a decade ago as a viable alternative to traditional TP-monitors in certain scenarios, we believe that SODA not only provides an alternative to traditional application servers but also enables a new class of service centric applications.

6. Acknowledgements

Our thanks to Pablo Castro, Brian Deen, James Hamilton, Gerald Hinson, Christian Kleinerman, Amrish Kumar, Srik Raghavan, and Florian Waas for feedback and corrections.

7. REFERENCES

[1] Helland, P. Data on the Outside versus Data on the Inside. *Proceedings of the 2005 CIDR Conference*

[2] Gray, J. Queues are Databases. *Proceedings 7th High Performance Transaction Processing Workshop*. Asilomar CA, Sept 1995.

[3] Blakeley, J. et. al. Hosting the .NET Runtime in Microsoft SQL Server. *SIGMOD Conference 2004*: 860-865.