The FactBase™: Robust and Secure Data Storage for Distributed and Disconnected Systems

Mike Lopez
Manukau Institute of Technology
Manukau, NZ
mike.lopez@manukau.ac.nz

To provide a positive user experience, an application generally needs to provide the user with a sense of the current state of the system. To support disconnected operation, data must be stored locally in caches. These rapidly become stale, leading to a reconciliation problem. Even when connected, getting agreement between distributed nodes on what constitutes current state is far from trivial.
Most applications use some sort of database to hold current system state, and from this perspective the problem of agreeing on current state becomes a distributed database problem. Reconciliation involves substantial programming effort and use cases proliferate with the need to program for both connected and disconnected modes of operation.
This paper introduces the concept of a FactBase™ which is a secure and robust distributed collection of Facts. Facts are defined as immutable aggregates of data. This immutability is the key to avoiding reconciliation issues and use case proliferation. A single programming model can be used for both connected and disconnected modes with the software moving seamlessly between the modes as connectivity is gained and lost.

1. INTRODUCTION

This paper starts by briefly reviewing current practice in database design and implementation and identifying the problems that are introduced by distributed and disconnected systems. It then demonstrates on a theoretical basis how these problems are a direct consequence of the approach we take. It raises the question of whether attempting to have a global current state is a useful concept and then proposes a new way of looking at the distributed data storage problem.

2. CURRENT PRACTICE

Much of our Systems Analysis has led us to analyse the data in our problem domain, eliminate duplication and redundancy, and define a central database containing normalised data. The database typically contains our current view of the problem domain and is implemented on a server. Transactions are used to promote "acidity" (Microsoft 2004). Although the concept of a central repository of data introduces a single point of failure, in practice this server-centric approach works well on local area networks.

When we distribute the application over a wide area network using a central server, the user experience is degraded by the lower communications speed and scalability (Microsoft 2004) is rapidly degraded by the use of transactions. This is because the transaction holds server isolation locks for an extended period of time.

In general, we have two options to moderate this degradation; we can replicate (Thompson 1997, Mysql 2004) the database at each local node and attempt to reconcile the diverging states from time to time; or we can use the central database and attempt to cache sufficient local data to enhance the performance and user experience and use server based processing, such as stored procedures, to limit transaction duration

When the system includes sometimes-connected nodes such as notebooks or handheld devices we generally have to provide for operation in both connected and disconnected modes. To do this, we have to cache sufficient data on the local device to provide a meaningful user experience while disconnected and then have to reconcile this local data with the on-line state when next connected.

Business-to-Business applications generally preclude the use of a single server since each business is unlikely to trust its sensitive data to a business partner. In consequence, these applications typically require the use of a distributed transaction coordi-

nator despite the scalability issues this brings (Microsoft 2004).

When we want to use a single architecture to develop an application that embraces all of this functionality we are really left with just one option; an approach that comprises holding local caches of data and the ability to reconcile the consequent differing views of current state.

Local caches of data are required to improve the user experience but deciding just what data to cache is by no means trivial. Even if adequate storage exists, caching a complete database on devices such as handheld or laptop computers in unlikely to be acceptable commercially for security reasons. Really, we need to use a different schema that limits the data stored on a need to know basis and this schema needs to be able to define content as well as structure.

Having decided to store caches of data at many locations, we then have the problem of ensuring that all the data stored is consistent. The most common technique is for each node to attempt to reconcile its local data with a "master" copy of data kept on some server.

For disconnected nodes, reconciliation use cases proliferate in a non-linear manner as use cases are added to the node and to the overall application. (In theory, if a node implements a subset of n use cases out of a total of m for the application, there could be at least $2n + n.m!$ use cases for the programmer to explore.) This use case proliferation means that it is not practical for the programmer to code each one, except for the most trivial applications. The result of this is a lack of robustness and a degraded user experience.

The underlying assumption of this reconciliation process is that the server holds the "current" state of the problem domain. Although theoretically flawed in all cases, this concept is generally useful on local area network based system, but its usefulness degrades as latency increases with wide-area and disconnected systems. In particular, there may be valid updates stored on some disconnected device and the server will only becomes aware of these at some future time when synchronization occurs.

3. THE PROBLEM

One part of the problem, thus, is the notion of global current state. In reality, whatever technology is used, each node has its own view of state that excludes work at other nodes that is in progress, or has been completed but not yet incorporated into global state. Attempting to resolve a global current state requires substantial programming effort and adds no value to the user experience at each node. In contrast, global historical state is easy to derive and has significant value.

The other part of the problem is that our databases are typically full of derived information (or state) rather than raw data. Whenever we store derived information then, by definition, we rely on some underlying process that derives this from raw data.

To reconcile diverging views of derived state, we need to know both the process that is applied and the underlying raw data that the process used. If we know both of these then we only need to store the raw data in the database.

3.1 The requirement

To solve the above problems, the key requirement is to eliminate all forms of derived data from the shared database and to implement an architecture in which current state is derived locally from the shared data to enhance the user experience. Global state can be made available on an historical basis.

3.2 The solution

In order to make a distinction between the suggested approach and current practice we will introduce the term Fact for raw data, and FactBase™ for its management software.

A Fact is defined as an arbitrary aggregate of immutable data. A fact is therefore inherently replicable and repeatable. In distributed application terminology we can say that a fact is idempotent (Fielding *et al.*, 1999). These characteristics guarantee that reconciliation of facts across nodes is trivial and can be automated.

Context is required in order to derive meaning from a fact and we define a fact context as the set of other facts that must be known in order for the current fact to be understood. Facts therefore naturally form a directed acyclic graph. This graph allows facts

to be processed correctly even when they arrive out of sequence.

The suggested approach is that applications can use a FactBase™ that has collections of facts stored at multiple nodes. Each node records the facts it originates and publishes those facts to interested subscribers. The node also caches copies of facts originating from other nodes. It can recover any lost facts from the original publisher, or its own lost facts from trusted subscribers. Fact storage, publication and subscription are easily automated in a standard manner.

The node can then use the facts to derive a logical object model that is stored locally. The logical object model represents the local view of the problem domain state and can be used to enhance the user experience. There is no need for the state of these local object models to be identical, only the facts need be shared.

No servers are needed. The approach eliminates single points of failure and promotes robustness by enabling massive redundancy.

The model works equally well in connected and disconnected states without use case proliferation. Rather than programming for two modes, all programming is carried out for a disconnected mode of operation. Each node simply records the facts it originates in the local cache and receives inbound facts from the nodes to which it subscribes. The only difference between connected and disconnected modes is the response time from other nodes. Fact publishing, replication and reconciliation is trivial and can be carried out automatically by standard software.

To help visualize the concept of what is meant by a fact, it is useful to consider an Audit Trail. By definition, an audit trail should document all changes to a system's state. It should also enable easy re-creation of the state. Above all, the audit trail must be immutable. Clearly, there is a close correspondence between the concept of an Audit Trail and that of a Fact.

4. VALIDITY

When evaluating storage technology we usually ask whether it passes the "Acid" test.

Atomicity: A fact is the unit of atomicity and is the analogue of a transaction. Facts arrive at nodes intact and complete, or do not arrive at all and can be any arbitrary aggregate of data.

Consistency: Derived state is updated only by the local node. The node can use standard synchronisation mechanisms to prevent access by other application threads while the state is being updated. These synchronization locks do not affect other nodes so there is no scalability problem.

Isolation: There is complete isolation of derived state from other nodes because the object model is local.

Durability: Copies of facts are kept at other nodes. This redundancy enables data to be recovered from other nodes without the need for central storage.

5. IMPLEMENTATION

To illustrate the concept, a proof of concept software package has been written for the basic FactBase™ with implementations for the dot net framework on the Windows desktop and the compact dot net framework on the Pocket Pc.

6. CONCLUSIONS

A fact-based architecture invites us to look at distributed applications from a fresh viewpoint. It also suggests an alternate methodology for analysis and design. The proposed architecture can be applied simply to all studied design patterns. For example, the allocation pattern requires the requester to publish a request fact and the allocator to publish a response.

The approach enables a single set of use cases to be used for both connected and disconnected modes. Performance is high when connected, degrading gracefully when disconnected.

The architecture allows distributed data storage. There is no need for a central point of reference, nor is there any benefit in having one. Robustness is enhanced through redundancy.

The approach taken in this paper suggests that the audit trail should take a central role in system design.

6.1 Future work

This work is part of an on-going project in distributed program architecture. Current work includes

the creation of a commercial strength implementation of the software.

REFERENCES

Microsoft, 2004, "Introduction to Transactions" Downloaded on May 12th 2004 fromhttp://www.sqlteam.com/item.asp?ItemID=15583

Microsoft, 2004, "Scalability" Downloaded on May 12th 2004 from http://msdn.microsoft.com/library/en-us/vsent7/html/vxconScalability.asp

MySql, 2004, "Replication in MySQL"Downloaded on May 12th 2004 from http://dev.mysql.com/doc/mysql/en/Replication.html

Thompson, c, 1997, "Database Replication"Downloaded on May 12th 2004 from http://www.dbmsmag.com/9705d15.html

Microsoft, 2004, "DTC Developers Guide"Downloaded on May 12th 2004 from http://msdn.microsoft.com/library/en-us/cossdk/htm/pgdtc_dev_0ulh.asp

Fielding et al, 1999, "Idempotent methods", published in rfc2616, 9.1.2 Downloaded on May 12th 2004 from http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html