Introduction

This is a short essay on some basics of information systems engineering. The topics covered are:

- Listener Dispatcher Handler: the fundamental middleware pattern
- ACID - Two Phase Commit: the fundamental capability of application middleware and resource management
- Transactional Messaging: the fundamental capability of messaging middleware
- Transactional Adapters: the fundamental capability of integration middleware, nested transactions
- BASE: the fundamental capability of master data management
- Normalisation: the fundamental capability of record keeping database design
- Facts and Dimensions: the fundamental capability of reporting database design

These are the fundamental engineering concepts an Enterprise or Solution Architect needs to keep in mind when creating solutions for information systems. These concepts form the constraints to information systems architecture in the same way that structural engineering concepts form the constraints to building architecture.

Listener Dispatcher Handler

All middleware, with the exception of agents which arguably are end-ware rather than middleware, has to be able to listen for events. Over the years since 1969, when IBM CICS was announced and it could be said that middleware was born, all middleware software systems have implemented the same basic pattern, which is illustrated below.

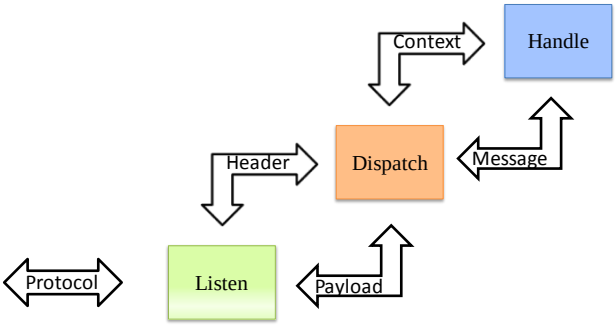

Figure 1 Listener Dispatcher Handler

The bottom level of the middleware is a listener. This listens for an event on a protocol. When an event is detected, the listener's job is to start the processing of the event in a thread of work. There are several different models for the way listeners do threading, of which the two main ones are: the TCP model, which is synchronous; and the MQ model, which is asynchronous.

In the TCP model the middleware listens then spawns. That is, it listens for a connection request on a well known port, and when it gets one it spawns a new thread on a banal port (or forks a process, depending one whether you are multiprocessing or multitasking), passes it the connection, and then goes back to listening. The activity diagram below illustrates the TCP listener.

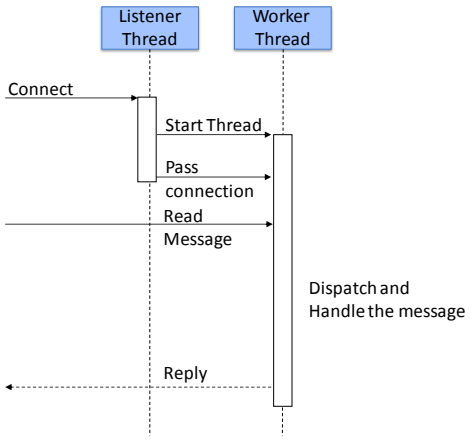

Figure 2 TCP Listener

In the MQ model, the middleware spawns then listens. That is, when the middleware starts, it spawns as many threads as there are queues to manage and then starts a listener for each queue. The gross

difference between the two models explains why much synchronous middleware (Tomcat, Apache, IIS and so on) does not support queuing very well. The MQ listener is illustrated below.

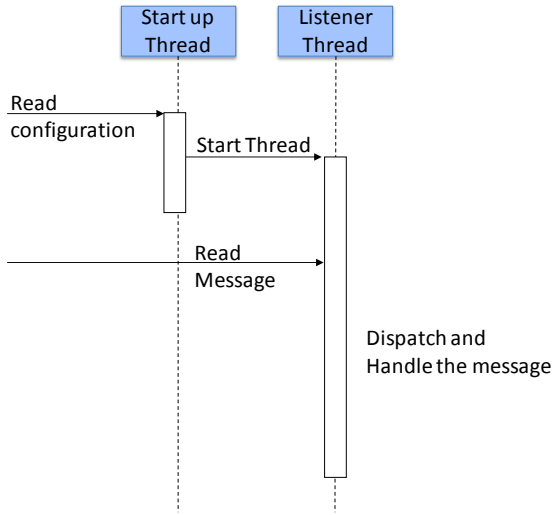

Figure 3 MQ Listener

The first work usually done on the worker thread is to dispatch the message. The reason why this is not always the first work is that some middleware runs cryptography after listening but before dispatching. The cryptography can decrypt the message if all or part of it is encrypted and can verify a signature by taking a message digest and using a public key to decrypt a signature token. Similarly, some middleware, notably on IBM mainframes, authenticates before dispatching, which is useful for minimising the impact of denial of service attacks. After decryption, the message may also need decompression. Then the header of the message is stripped from its payload and the headers and payload are sent to the dispatcher. The job of the dispatcher is to choose which handler to run for the event. In Web middleware the dispatcher uses the HTTP headers, the URL and the MIME Type to determine which handler to run. In Ruby on Rails the process of working out which handler to run for a URL is called 'routing'. In our terminology, the routing is done by a dispatcher.

The dispatcher takes all the information known about the message, including the protocol headers and any side information from the configuration and creates a context for the handler. It passes this context and the payload of the message to the handler. Some middleware, just before passing the payload to the handler, maps the content of the payload. For instance, both IMS and CICS on mainframes map messages from 'green screens' from their native data stream into the segment for the handler. When sending messages from CICS to CICS there is an optional transform in and out. Similarly, XML middleware can use XSL-T to transform the message into another format suitable for the handler, such as text or HTML. In Java Enterprise Edition, JAX-B can transform the message to native Java objects. This enables the handler to be independent of the form of the message 'on the wire'.

The handler is responsible for doing what is needed to manage the message. If the handler is an application it will be stateful, that is read and write persistent storage, and will probably start by correlating the message to some stored state. For instance, if the handler is orchestrating other services, it will need to look up the state of the workflow using keys in the message, and when it has completed it will need to update the state before committing.

Application middleware is designed to reply synchronously to the message received, as illustrated below.

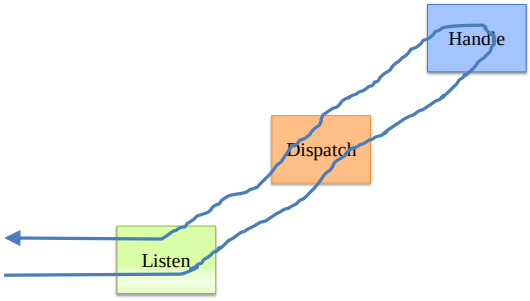

Figure 4 Synchronous Handling

Integration Middleware

Integration middleware has to act as a relay. In general, this will be asynchronous, as shown below which is

how, for example, a JMS message would be handled.

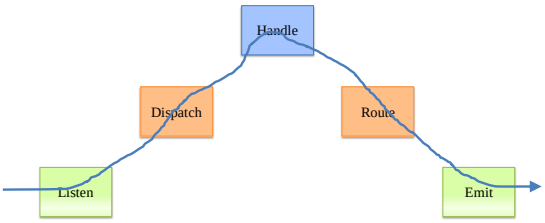

Figure 5 Asynchronous Handling

In order to act as a relay, the middleware has to add a router, to decide on which protocol to emit the message, and an emitter to act as a protocol client.

When the protocol being listened on is synchronous but the protocol being emitted on is asynchronous, for instance if the request comes in on HTTP but the consequence of the request is sent on MQ, then the pattern used is as shown below.

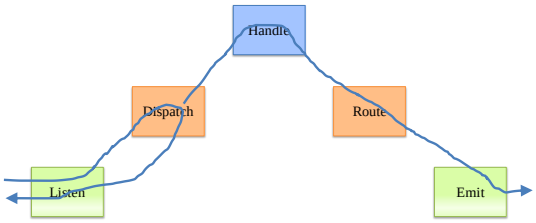

Figure 6 Synchronous Listen with Asynchronous Emit

When application middleware runs this pattern, the listen and emit are part of a single transaction and the transaction either commits or rolls back. This relies on there being a person as the requester who decides whether to resubmit the transaction or not. However, when dealing with a message from another system, there is no person to manage the transaction. In this case there are effectively two transactions going on in the relay. The first transaction listens for a request and replies when done. The second transaction, nested inside the first, emits the handled message. This is very important to grasp. In handling messages as a relay, we do not rely on a two phase commit. Rather we use a nested transaction, each leg of which may itself be a two phase commit. The reason the whole relay cannot be a two phase commit is that the relay has to take control of the message. There is no person at the requesting end to manage the transaction. If we used a two phase transaction then if the handler failed, the message would roll back. The listener would then retry the message and the handler would fail again. Eventually, the message would exceed its roll back limit and would go to the error output. This would be nearly useless as the message would have no information on what went wrong. Instead, if the nested transaction fails, the outer transaction can decide whether to retry, in which case the message goes to a retry location, or to error, in which case the message, with the information about the problem, goes to the error location. Then the outer transaction can commit and go on.

If the handler needs to run a flow to manage the message, for instance if it is using SAP remote function calls to apply the received message to an SAP system, then it may need to emit several times as shown below.

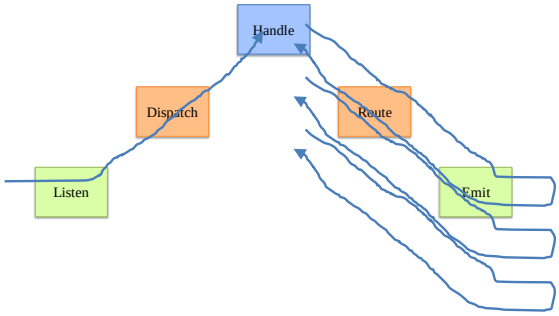

Figure 7 Complex Handling

If the listener is for MQ and the emitter is for SAP then the both the listen transaction and the emit transaction can be two phase. This enables the complex handler to run successfully and atomically.

Two Phase Commit

The concept of a two phase commit was invented during the 1970s as transaction monitors were being developed at IBM and Tandem. Jim Gray was the principal researcher and implementer as he was involved in the development of both IMS and System R (the first distributed relational database). We can do no better than to quote his description of a two

phase commit.

"It is generally desirable to allow each participant in a transaction to unilaterally abort the transaction prior to the commit. If this happens, all other participants must also abort. The two-phase commit protocol is intended to minimize the time during which a node is not allowed to unilaterally abort a transaction. It is very similar to the wedding ceremony in which the minister asks "Do you?" and the participants say "I do" (or "No way!") and then the minister says "I now pronounce you", or "The deal is off". At commit, the two-phase commit protocol gets agreement from each participant that the transaction is prepared to commit. The participant abdicates the right to unilaterally abort once it says "I do" to the prepare request. If all agree to commit, then the commit coordinator broadcasts the commit message. If unanimous consent is not achieved, the transaction aborts. Many variations on this protocol are known (and probably many more will be published)."

When doing something in response to a business event our handlers may have more than one resource to manage. For instance, the message may have been received on a queue and the handler may need to write to a database. In this case, the queue and the database must be coordinated to do all of their actions or none of their actions. This is what the two phase commit, in combination with a log, allows us to ensure. The alternative to the two phase commit would be to rely on complex programming in every handler. In other words, the transaction is a complex, cross cutting concern.

Here is how it works. The model for two phase commit transactions was standardised by the Open Group when it was still called X/Open. This model is illustrated below.

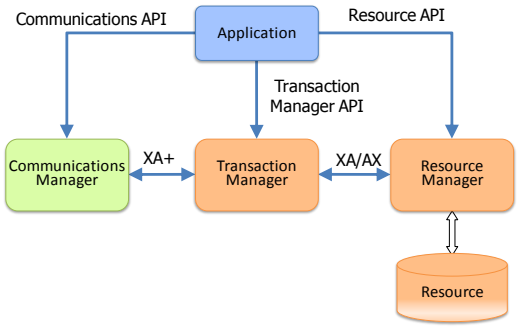

Figure 8 X/Open Transaction Model

The actors in the diagram are: the application program; the transaction manager; the resource manager; and the communication manager. If there is just one resource manager and no transaction or communication manager, then the transaction is described as being local. This would be the case when using SQL START WORK to delimit SQL transactions. If there in one transaction manager and one or more resource managers, then the transaction is global (even if the resource managers are themselves running on other machines). If there is more than one communication manager then the transaction is distributed. X/Open proposed three standards. XA is the standard for a transaction manager to include a resource manager in a transaction. This is implemented in Java Extended Edition. AX is the standard for a resource manager to include itself in a transaction (dynamic registration). This is not part of Java EE. Finally, XA+ is the standard for distributing transactions. X/Open never completed this standardisation (it is just too difficult). However, Java EE does have an implementation of this, though you would probably be mad to use it. So, in effect, only XA is important. This is why Java transactions are often called XA transactions.

There are also three APIs: the API to the resource (SQL or JMS say); the API to the transaction (JTA in Java EE); and the API to the communications manager. Note that in the latest version of Java EE, all three of these can be container managed. SQL if you use container managed persistence,

JMS if you use a message driven bean, and JTA if you use container managed transactions.

In the case of a Java EE transaction, you have to configure the EJB to use transactions and register the resource managers to use them before the EJB executes. When the EJB is executed a transaction is created and logged. When the EJB uses the API to a resource, that resource manager is added to the transaction and its events logged. When the transaction commits, the transaction manager runs the first phase, prepare. It asks each resource manager to prepare. Once an RM has agreed to prepare it cannot abort the transaction, which means that the RM must log its important events so that they can be read later for undo (roll back) or redo (commit). If any RM refuses to prepare, the transaction is rolled back. Once all RMs have prepared, the TM starts the second phase, commit. Each RM is committed in turn. The time between issuing 'prepare' and then issuing 'commit' or 'roll back' is known as the 'in-doubt window'. During this time the RMs are in doubt about whether to commit or roll back. Finally, the TM logs the commit. If there are any actions that cannot be undone, they are done by the RM at commit. In particular, message PUTs are not actioned until commit.

The attributes of a global transaction are that it is ACID. This is a term coined in 1983 by Andreas Reuter, here is Jim Gray's definitions of ACID.

"Atomicity: A state transition is said to be atomic if it appears to jump from the initial state to the result state without any observable intermediate states—or if it appears as though it had never left the initial state. It holds whether the transaction, the entire application, the operating system, or other components function normally, function abnormally, or crash. For a transaction to be atomic, it must behave atomically to any outside ''observer".

Consistency: A transaction produces consistent results only; otherwise it aborts. A result is consistent if the new state of the database fulfills all the consistency constraints of the application; that is, if the program has functioned according to specification.

Isolation: Isolation means that a program running under transaction protection must behave exactly as it would in single-user mode. That does not mean transactions cannot share data objects. The definition of isolation is based on observable behavior from the outside, rather than on what is going on inside.

Durability**:** Durability requires that results of transactions having completed successfully must not be forgotten by the system; from its perspective, they have become a part of reality. Put the other way around, this means that once the system has acknowledged the execution of a transaction, it must be able to reestablish its results after any type of subsequent failure, whether caused by the user, the environment, or the hardware components."

The combination of a log, locking and a two phase commit ensures ACID properties.

Transactional Messaging

An example of using transactions is the use of transactional messaging. Messaging and Queuing uses PUT and GET verbs. A PUT sends a message, a GET reads a message. Both are changes to queues as GET is a destructive read in M&Q. A typical message driven bean GETs from one queue, updates a database and then PUTs to another queue. This is illustrated below.

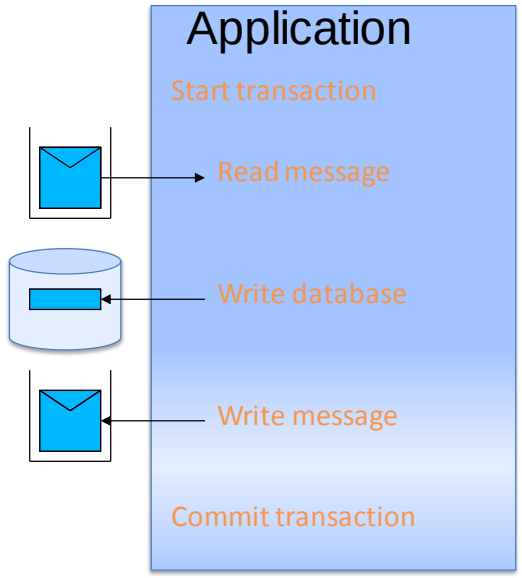

Figure 9 Transactional Messaging

The application first GETs a message, then writes to a database, then PUTs a message and finally commits. If the transaction were to roll back instead, then the message is put back into the input queue, the database is rolled back and the message in the PUT queue is removed.

A scenario that we encounter quite often is that an application that was used directly by a person using an agent is changed so that it is now accessed via another record keeping application. Many designers fail to notice that this now implies a distributed transaction as illustrated below.

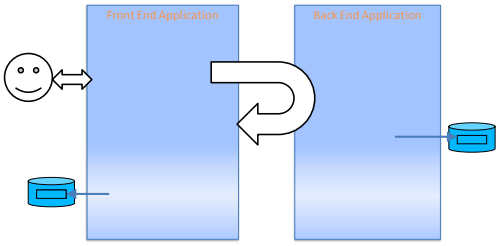

Figure 10 Accidental Distributed Transaction

The front end application now has its own resource, shown as a drum, as well as the resource in the back end. Therefore, a synchronous request from the front end to the back end implies a distributed transaction. This does not scale as the two phase commit blocks and requires a session to remain in place until commit.

The right way to implement this is as shown below with three transactions, not one.

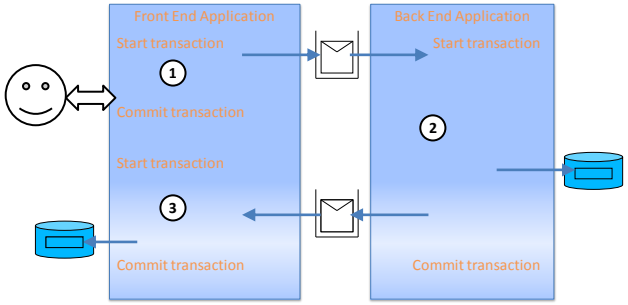

Figure 11 Three Transaction Model

The way to do this is to use three transactions to make the change in the back end application. The user makes a request that causes a change to the front end application. While holding the user session, the front end puts a message to the back end queue and commits. The back end GETs from the queue, updates the database and PUTs to the front end queue (note that this is not a reply-to message). The front end listens on the queue, reads the database to get back to the user session and completes the request.

BASE

The target enterprise architecture of the 1980s was to integrate through the database and to use distributed transactions to update a database across a set of nodes. This architecture failed for several reasons. Firstly, it didn't scale. The application processing bottlenecked on the database. Secondly, it was not possible to update through views and so applications were too closely coupled. Finally, the distributed transaction was too slow and too unreliable (as you scale up distributed transactions, more and more of them fail to commit). IBM created transactional messaging as a way of distributing transactions instead of two phase commit and message passing replaced a shared database as the approach to integration. No longer did the architecture rely on ACID transactions for integration (though it still requires global ACID transactions so that databases and queues can be used

in transactions). The replacement for ACID is called BASE and stands for Basically Available, Soft state, Eventually consistent). At the heart of this approach is the CAP theorem. This states that of the three desirable properties of an information system, consistency, availability and tolerance to partition, you can at most have two. As we are doomed to have partitions, this leaves us with a choice between availability and consistency. As availability is usually a must have for enterprise systems, that means we have to live with a compromised consistency. The way we do this is to use business event sharing to make each information system autonomous and asynchronous. When a master data change is made in one system, it is propagated to all other interested systems either by a hub (within a domain) or by agreements (across domains).

REST

The approach we take to designing applications uses the architecture of the Web, Representational State Transfer, or REST for short. REST has four principles:

1. All resources are addressed with a URL
2. All resources have a uniform interface. In standard HTTP this is GET, PUT POST where GET is safe, PUT is idempotent and POST is neither.
3. All resources have multiple representations
4. The application (user interaction part) has its state driven by the state of the current resource. This is known as Hypertext as the engine of application state or HATEOS. Although this third part is the least well implemented, it is probably the single most important part.

Change Verified Protocol

When applying changes to resources, it may be necessary to manage concurrency yourself. This is the case if there are competing transactions for the same resource (humans and queues for example). The way this works is that the change is applied in two phases. The agent making the change has to have the changed values it wants to apply and also the previous values it found in the resource before it started the change. In the first phase the object to be changed is read. This can be done as part of a transaction with ‘repeatable read’ configured as the level of concurrency. This locks the resource as part of the transaction. The values read are then compared to the before values of the object that the agent has kept. If these are the same it is safe to apply the change. If they are not the same then the object has been changed before this change has been applied and the agent needs to start the race again.

Syntax and Semantics

An awful lot of what we talk about in architecture seems to end up being about distinguishing syntax from semantics. Because of that, here is a short introduction to what is usually meant by these two terms. However, be aware that there is more than one definition of these terms and the definition in use can change in other contexts. Also, depending on context, one person’s semantics might be another person’s syntax.

Generally speaking, syntax is about the form of things and semantics is about their meaning. Examples of syntax include XML which has rules for what it is to be well formed. These rules are purely syntactic. The fact that the tags in XML are always delimited by angled brackets adds nothing to the meaning of those tags. Indeed, XML evolved from SGML which evolved from GML and in GML tags were delimited by colons and dots. So <p> in XML was :p. in GML. If a message

or file or data stream is converted from XML to GML or JSON or some other syntax, nothing changes in the meaning of the message.

Semantics, however, is about the meaning of the things you are dealing with. In a relational database these are relations and tuples (or tables, rows and columns). In a database model these are entities, relationships and attributes. In a message these are data elements and attributes. In a relational database the possible values of a column are known as its domain. Specifying the domains of the elements of a relationship is a semantic specification. Some people get confused over domains and think of them as syntactic. This is because when a domain is specified it is necessary to give it some syntactic representation. If the attribute is Gender and we say it can be 'M' or 'F' then the meaning inherent to the domain is semantic but the choice of 'M' and 'F' rather than '0' and '1' or 'H' and 'F' (which a French modeller might choose) is syntactic.

In the world of XML, a W3C XML Schema is semantic as it provides the structure of the document and the domains of its elements. Similarly an XSL-T transform from one schema to another is also a semantic thing as it equates what something means in one message with what it means in another. Semantic modelling is the term used for data base modelling using entities, attributes and relationships.

Normalisation

Normalisation is as old a concept as the relational database. In his first paper on the relational data model, Ted Codd noted the importance of normalisation in order to remove update anomalies. This section on normalisation is included as it is an often misunderstood concept in the design of databases.

The important thing to know is that normalisation is not a way of discovering business rules. In fact, the opposite is the case, you need to know the business rules in order to do the normalisation. Academics describe five normal forms (first, second, third, fourth or Boyce-Codd and fifth). They are progressive, in that if you are in say fourth normal form then you are also in third, second and first normal form. Also, all of them are about dealing with 'dependencies', that is, if John is the child then Seymour is the father (the choice of John determines Seymour) or, conversely, you cannot have the child unless there is a father.

In most cases, if you have a model in third normal form then it is likely to be in fourth and fifth normal form unless there are some many to many relationships involved. This is because third normal form removes functional dependencies, whereas fourth and fifth removes many valued dependencies. For instance, where you have students taking courses, faculties offering courses and teachers employed in faculties. There are many to many relationships between students and faculties and students and teachers and students and courses. In general, eliminating dependencies also eliminates redundant data, though some redundant data may always be present even in fully normalised relations. This is not the primary aim of normalisation, but does help with maintenance, scaling and integrity.

If normalisation is about removing update anomalies it is probably a good idea to understand what they are. An update can be an insert, a change or a delete. If I had a table of employees, their addresses and their skills, with a row for each skill, then changing the address has an update anomaly because I might forget to change one row leaving the address inconsistent. If the employee table has the employee id, name and project, then we cannot add an employee who is not

yet assigned to a project, this is an insert anomaly. If an employee finishes one project but has not yet started another, then deleting the current row also deletes our knowledge of the employee and their address, which is a delete anomaly.

Briefly, then, here are the normal forms. A relation is in first normal form if each attribute only has one value. In a table of books, there can only be one value for the author field and the field is not repeated. To record that a department has more than one employee you need to repeat the book row. A relation is in second normal form if it is in first normal form and each attribute depends on the whole of each key. In a table of employees at locations, the address of their location only depends on the location part of the key. You can say that first normal form is about redundancy of data across a row, second normal form is about redundancy of data down a column.

A relation is in third normal form if it is in second normal form and each attribute is independent. This means that there are no transitive dependencies, which in turn means that there are no functional dependencies. In a table of employees with departments, the department is functionally determined by the employee key (an employee has one department) but the department location is not determined by the employee key except through the department key.

It used to be that we thought there was one data model for the enterprise and, if it was normalised, it represented at some level, the business rules for the enterprise. But this failed to acknowledge the importance of separating the events part of the enterprise information system from the content part of the enterprise information system. In the content part there are no updates and so no update anomalies. Therefore normalisation is not required, in fact it is strongly deprecated. We gradually learned that the content part of the enterprise information system is modelling a completely different aspect of the enterprise.

Facts and Dimensions

In the events part of the enterprise information system we are concerned with managing events and storing enough information about the events that we can process the next event. This requires knowing the pre-condition and post-condition for the event handling. Twinkling data bases are those that are constantly changing as they handle events. Each change of the data base must take it from a consistent state to a new consistent state. The data base must be immune if possible from update anomalies. The state of the data in the data base must represent a set of true statements about the enterprise. The way we do this today is to use normalised relational data bases described by entity attribute relationship semantic models.

However, we no longer consider the data model to be universal. That is, we do not pretend to have a single model for the enterprise. This is for two reasons. We discovered it was not possible to scale a single data base to record all events. The Sysplex project I worked at when at IBM in the late 80s was created because of the database scaling problem. More significantly, we also discovered that one of Ted Codd's conditions for the relational model, that it must be possible to update using the views, was not achievable with the technology at hand. In fact, increasingly, it looks as though it will never be possible to update through views. This made it impossible to integrate different applications through the database which in turn

lead to a new way of integrating, through messaging. Again, this was why we created transactional messaging at IBM in the late 80s.

If normalised relational databases are the norm for events, what is the equivalent for content? And what about the enterprise are we modelling in the content database? It seems that the right model for content is the fact and dimension model where we have normalised fact tables consisting of two parts. The first part is a set of facts, such as the quantity and the price of transactions that have been recorded. The second part is a set of keys relating to dimensions that can be used to slice and dice the facts. Each dimension represents a role played by a party to the transaction. Typical dimensions are buyer, selling, agent, product, place (political and physical), time, line of business, contract terms and business function. The dimensions are made up of master and reference data whereas the facts are made up of transaction data.

What we are modelling in such a database is typically a process represented by each fact table. Processes, in English, are usually words that end with …ing (gerunds in grammatical terms). Examples include selling, buying, hiring, incurring risk, accounting and so on. Unlike normalised databases which are modelled using entities attributes and relationships, multi-dimensional databases are properly modelled using a state machine. The business events represent transitions in the state machine. The fact tables record all the transitions. The dimensions enable us to slice and dice the process. With this model, it is possible to answer any conceivable question about a process.

The details of multi-dimensional model occupy many books. But very few tell us how to relate the state model to the fact table. Getting that right is a major research aim of our EA practice.

About John Schlesinger

John Schlesinger is a Principal at Atos Consulting where he leads its Enterprise Architecture practice. John is an advisor to enterprises specialising in middleware and integration architecture. He has lead integration architecture development in retail banks, investment banks, retailers and manufacturing, both for integrating applications and for integrating information.

John has worked both as a consultant and also as a developer with software companies. He has taken over two dozen program products to market at IBM, Information Builders, One Meaning, SeeBeyond and iWay Software. These products included the world's most successful commercial software (CICS) and the world's most successful data middleware (EDA/SQL). John also led the Architecture department at Dun and Bradstreet when its IT department went global.

A member of the ACM and the IEEE, John has an MA in Physics and Philosophy from Oxford University and a Post Graduate Diploma in Software Engineering from Oxford University.

John has spoken at numerous conferences including the CIO Cruises run out of New York, during one of which he was the first speaker on after the collapse of the World Trade Towers in 2001.

John can be contacted at john.schlesinger@atosorigin.com