The VLDB Journal (1997) 6: 173–190

The VLDB Journal
© Springer-Verlag 1997

The impact of object technology on commercial transaction processing

Edward E. Cobb

IBM Corporation, Department DQGA/A230, 555 Bailey Ave., San José, CA 95141, USA
e-mail: ed_cobb@vnet.ibm.com

Edited by Andreas Reuter, Received February 1995 / Revised August 1995 / Accepted May 1996

Abstract. Businesses today are searching for information solutions that enable them to compete in the global marketplace. To minimize risk, these solutions must build on existing investments, permit the best technology to be applied to the problem, and be manageable. Object technology, with its promise of improved productivity and quality in application development, delivers these characteristics but, to date, its deployment in commercial business applications has been limited. One possible reason is the absence of the transaction paradigm, widely used in commercial environments and essential for reliable business applications. For object technology to be a serious contender in the construction of these solutions requires:

– technology for transactional objects. In December 1994, the Object Management Group adopted a specification for an object *transaction service* (OTS). The OTS specifies mechanisms for defining and manipulating transactions. Though derived from the X/Open distributed transaction processing model, OTS contains additional enhancements specifically designed for the object environment. Similar technology from Microsoft appeared at the end of 1995.
– methodologies for building new business systems from existing parts. Business process re-engineering is forcing businesses to improve their operations which bring products to market. *Workflow computing*, when used in conjunction with *"object wrappers"* provides tools to both define and track execution of business processes which leverage existing applications and infrastructure.
– an execution environment which satisfies the requirements of the operational needs of the business. Transaction processing (TP) monitor technology, though widely accepted for mainframe transaction processing, has yet to enjoy similar success in the client/server marketplace. Instead the database vendors, with their extensive tool suites, dominate. As object brokers mature they will require many of the functions of today's TP monitors. Marrying these two technologies can produce a robust execution environment which offers a superior alternative for building and deploying client/server applications.

Key words: Objects – transaction processing – Workflow

1 Introduction

Today's *transaction processing* (TP) systems provide an environment for executing applications which are critical to the *operational* needs of business. These systems have evolved over a 30-year period from mainframe-centric environments supporting fixed function terminals to today's client/server environments where application function is distributed among many cooperating processors.

The world's businesses have massive investments in the applications that make up these systems – estimates exceed a trillion lines of application code, and the business processes implemented by these systems are *"mission critical"*; they have no manual backup. The TP system **is** the business. If the system is not functional, orders go unprocessed, business is lost to competition, and even survival can be threatened.

A TP system contains the business rules and processes to collect and analyze data from business events and use that data to make operational decisions. The system software that enables these applications embodies the following components:

– A *transaction manager* which provides the **ACID** properties (**A**tomicity, **C**onsistency, **I**solation and **D**urability) essential to the integrity of the system.
– A lightweight scheduler which acts as a switch connecting a very large number of clients with a modest number of re-usable servers to process client requests.
– A set of administration facilities, including performance analysis and tuning, accounting, change management, and problem determination, which consistently manage the applications and data within its domain.

Today these functions are packaged in a monolithic software package, commonly referred to as a *TP monitor*. IBM's CICS, BEA's Tuxedo, and Transarc's ENCINA are examples of successful TP monitors. TP monitors, together with a database system and a set of application development tools, enable today's customers to build and operate mission critical commercial transaction processing applications.

Object technology will change that. The possibility of thousands of objects communicating with each other in ways not even imagined by their designers will allow businesses

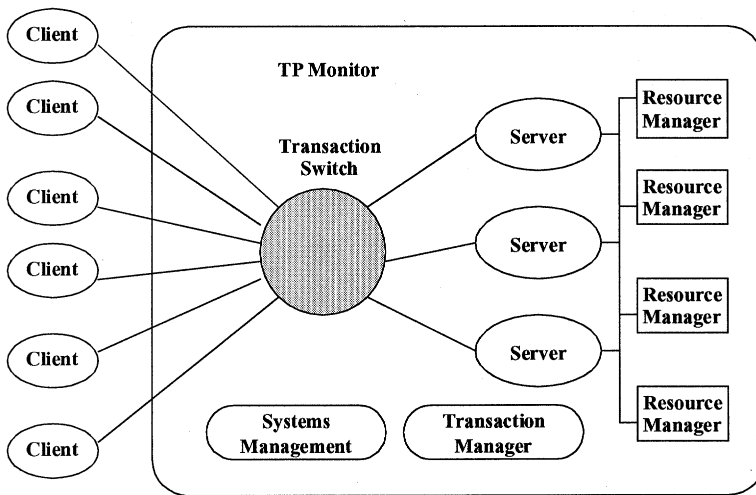

Fig. 1. A transaction processing system

to assemble more sophisticated applications more rapidly, enabling the business to react to new opportunities in the marketplace.

1.1 Customer environment

Since its beginning, information technology has been synonymous with change. Ledger books became card files, which became disk files and then databases. Today they are an *information resource*, vital to the operation of the company. Business environments have been changing as well. Local and national markets became global and competition is also world-wide. The sophistication of a company's information resource can make the difference between staying ahead or becoming a casualty of today's highly competitive environment.

Information technology (IT) systems are an integral part of a corporation's strategy to compete. Whether it is sophisticated reservations software to maximize revenue per seat or customized long-distance calling plans, today's information systems are a vital part of today's businesses. They form an electronic infrastructure, essential to daily operation and management of the enterprise, and enable new services not previously possible.

Computer-buying decisions based on hardware platform are a thing of the past. Today customers look first for applications to give them a competitive edge; then the requisite hardware and software. Applications are becoming more specialized, tailored to the requirements of a specific industry or even a niche within an industry. Application development must become more productive and responsive. New business opportunities cannot be exploited without a supporting information system. The time to develop a new application directly impacts the competitiveness of the corporation.

1.2 Industry trends

The rate of technological change continues to accelerate. At the same time, the cost of computing continues to plummet. Communications bandwidth that seemed inconceivable only a few years ago is taken for granted today. But technological change is more than just decreasing costs. New technologies make possible capabilities not previously envisioned.

1.2.1 Distributed computing

Distributed systems are critical to the future of computing. Competitiveness demands the maximum use of a company's information assets. Previously independent systems must be linked together whether they are within an enterprise or between enterprises. As people need to communicate, so do their business systems. Distributed computing is more than just cost-effective technology; it is the ultimate representation of business organization.

Increasingly, end-users interact with these systems through personal computers. PCs offer significant improvements in end-user interface technology. An airline reservations clerk can make seat assignments by viewing a graphical display of the aircraft and selects an empty seat by pointing at the desired location. This makes these applications easier to use and more intuitive.

Departmental servers make it possible for computations to be performed at multiple locations. For example, much of the information required to operate a warehouse, is in fact local to that warehouse. Departmental systems allow data to be easily shared among the users of the warehouse, independent of the corporate system and network, allowing response times to be faster and availability to be improved.

Inter-enterprise networking, *electronic data interchange (EDI)*, is showing explosive growth across all industries. Customers are connecting to suppliers. Funds are transferred electronically. Stocks and bonds are bought and sold at computer speeds. Customer engineers order parts with hand-held computers over cellular networks. Inter-enterprise networking is no longer the exception; it is the normal way of conducting business.

1.2.2 Middleware

Distribution leads to heterogeneity in both hardware and software. The number of IT suppliers continues to explode and the need for standards that allow disparate systems

to work together has never been greater. *Middleware*, the "glue" necessary to make this collection of piece parts work together in a cohesive manner, bridges disparate hardware and software environments to enable real business solutions.

Middleware comes in many flavors:

- communication software which permits the computers to talk to each other,
- data access software which manipulates data stored in databases from multiple vendors,
- messaging software which enables applications on different platforms to communicate, independent of the communications network,
- TP monitors which provide tools and an environment for deploying distributed applications,
- *Object request brokers (ORBs)* which provide infrastructure to support distributed object computing.
- Systems management tools which make it possible for the enterprise to cost-effectively manage a heterogeneous collection of hardware and software.

1.2.3 Systems management

Substantial pressure is being placed on IT professionals to support more users on increasingly more complex heterogeneous systems, often with fewer people. With computing power and data dispersed across the network, systems management becomes even more complex.

- Every system needs some form of configuration.
- Software requires corrective service which must be applied and distributed.
- Problem determination and performance management are far more complex.

Whether distribution occurs by deploying robust mainframe technology on the desktop or by evolving proven desktop solutions to support the enterprise, there are issues of scalability and manageability that dwarf the issues of the glasshouse environment of only a decade ago. Making these systems manageable, at reasonable cost, is the biggest challenge the industry faces to fully exploit the potential of distributed computing.

1.3 The promise of object technology

An important element of this middleware and one destined to have a major impact on information technology in the coming years is *objects*. Object technology holds the promise of significant improvements in programmer productivity and quality [14]. This promise is derived from the strong potential for re-use in the object properties of encapsulation, inheritance, substitution, and polymorphism. As a result:

- object technology is ideally suited to distributed processing.

  Encapsulation allows an object to be located anywhere in the distributed system. Locality based on today's performance and availability requirements can be changed administratively tomorrow as the business environment changes without changing the object's users. This contrasts markedly with today's procedural distribution techniques which require function placement decisions to be made at application design time and require significant application re-work to adapt to changing requirements.
- Object technology mirrors the enterprise by modelling its business processes.

  In an analysis of companies that have shown significant increases in productivity through the use of information technology, *Business Week* concluded that the key to achieving these benefits is coupling the use of technology directly to business engineering efforts [24]. This allows the information system to mirror the business infrastructure, eliminating translation errors, and making the system more adaptive to future change.
- Object technology encourages incremental adoption.

  With simple *wrappering* techniques, legacy applications can be easily integrated into new object-oriented applications based on the enterprise business model. This makes it an ideal strategy for building new applications to exploit new opportunities while leveraging the existing application infrastructure.
- Object technology provides independent software vendors (ISVs) the tools and techniques to bring new applications to market quickly.

  The *"software component industry"*, will be made possible by re-using the objects of existing applications in producing new ones, allowing applications to be assembled from existing parts and thereby constructed more rapidly. With widespread deployment in the industry, the software developer is offered a significant market, regardless of the underlying computer systems in use.

1.4 OMG's object management architecture

An architecture for distributed objects capable of communicating with each other across a myriad of network protocols is being developed by the *Object Management Group* (OMG). The *object management architecture* (OMA) defines the primary components of a distributed object environment available on each platform in the network. The overall architecture of OMA is depicted in Fig. 2.

The central component of the OMA is the ORB. The ORB enables objects to send messages to other objects [18]. The ORB provides location transparency and hides the details of communications.

CORBAservices is a collection of services with object interfaces that provide basic functions for realizing and maintaining objects [19]. Services to support object-naming, event notification, life cycle operations, and persistent storage of objects were adopted in 1993. Specifications for transactions, concurrency control, relationships, and externalization were adopted in 1994. In 1995, services for query, properties, and licensing were added and security and time services are imminent.

CORBAfacilities are high-level application services that provide uniform interfaces and semantics used to build well-formed applications in a distributed object environment and developed on a CORBA-conformant ORB. The OMG is

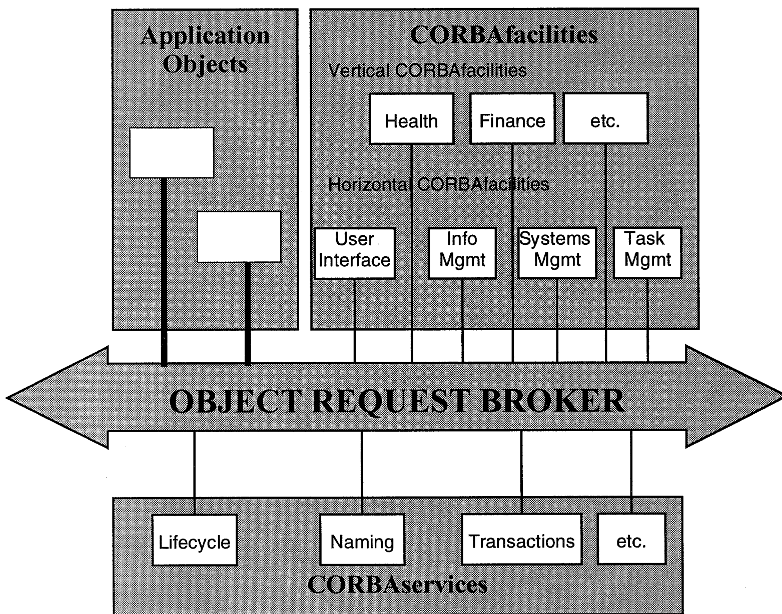

Fig. 2. Object management architecture overview

initially focusing on cross-industry requirements, including compound documents and systems management.

Application objects are specific to particular industry domains and are being addressed by vertical industry special interest groups (SIGs). In early 1996, the OMG re-organized to increase its emphasis on business objects in the application domain.

1.5 Why objects are ideal for transaction processing

To date, object technology has been most successfully deployed in environments which either have no need for transaction semantics (e.g. graphical end-user interfaces) or in application areas where traditional transaction technology is inadequate (e.g. engineering design). But this will change. Distributed objects can be used to construct client/server applications from self-managing components which interact with each other independent of network or operating system [22]. To realize these benefits in traditional OLTP environments, transaction technology must be added to objects.

– The OMG took this step in 1994 when it adopted the *Object Transaction Service* (OTS) specification for its object broker, CORBA.
– Microsoft released a specification (OLE/TP) for adding transactions to OLE and its object broker, COM, in late 1995.

So the stage is set. With the marriage of transactions and objects, object technology is poised to have a major impact on TP. This will happen for three reasons.

1. Reusable components are the key to rapid development of new business applications, and these applications require *transaction semantics* to support shared data access with integrity.
   Introducing these semantics to object technology is a critical component of any solution and must be done in a way which encourages widespread component re-use for building commercial applications. This is explored in Sect. 2.
2. Incremental enhancement of existing TP systems requires the re-use of legacy applications with new applications to support new business processes developed through today's re-engineering efforts.
   Business processes tend to be long running and involve people as well as programs. Today's TP systems understand only short-duration transactions and have limited interactions with people. To support long-running business processes, a mechanism for specifying and recovering processes as well as data is required. This is explored in Sect. 3.
3. Many commercial applications are deployed in TP monitor environments.
   Is ORB technology compatible with TP monitors? Are TP monitors even relevant in this new environment? If so, what changes are necessary to make them work well together? This is explored in Sect. 4.

2 Transactional applications with objects

The transaction [6] concept is an important programming paradigm for simplifying the construction of reliable business applications. First deployed in commercial applications, where it was used to control shared access to data in centralized databases, it has more recently been extended to the broader context of distributed computing. Today, it is widely accepted that transactions are the key to constructing reliable distributed applications.

2.1 The transaction paradigm

The term, *"transaction"* is often used rather loosely, but there are a multiplicity of *transaction models*. Each model defines a set of rules for providing the ACID transaction

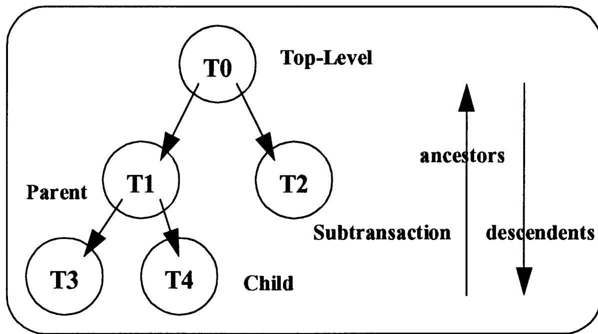

Fig. 3. A transaction hierarchy

properties. The most commonly used models are described briefly below.[1]

2.1.1 Flat transactions

The flat transaction model provides the mechanisms for relating a set of computations which demonstrate the ACID properties. All accesses to *recoverable* data within the scope of a transaction exhibit

- *atomicity* – the set of computations is either completely done (committed) or completely undone (rolled back).
  The state of all recoverable data changes uniformly and only at transaction boundaries.
- *consistency* – the effects of a transaction preserve invariant properties.
  The state of recoverable data is visible to other applications only when committed.
- *isolation* – intermediate states are not visible to other transactions.
  Transactions appear to execute serially, even if they are performed concurrently.
- *durability* – the effects of a completed transaction are persistent.
  Changes to recoverable data survive even in the event of system failure.

The flat transaction model is widely supported in the industry. Implemented by TP monitors, databases, files, and queuing systems – flat transactions are the basis for the X/Open Distributed Transaction Processing (DTP) model [26] and the ISO OSI-TP standard [10].

2.1.2 Nested transactions

Nested transactions [16] allow the creation of transactions embedded within an existing transaction to form a transaction hierarchy (Fig. 3). The existing transaction is called the *parent* of the embedded transaction. The embedded transaction is a *subtransaction* and is referred to as a *child* of the parent transaction.

Subtransactions can be embedded in other subtransactions to any level of nesting. The *ancestors* of a transaction are the parent of the subtransaction and the parents of its ancestors. The *descendants* of a transaction are the children of the transaction and the children of its descendants. Subtransactions are strictly nested. A transaction cannot commit unless all of its children have committed. When a transaction is rolled back, all of its children are rolled back.

A *top-level* transaction is a transaction without a parent. A flat transaction can be thought of as a childless top-level transaction. A top-level transaction and all of its descendants make up a *transaction family*.

Subtransactions are atomic. However, when a subtransaction commits, its changes remain pending until commitment of all its ancestors. Thus subtransactions are not durable – only top-level transactions are durable.

Isolation applies to subtransactions. When a transaction has multiple children, a child appears to execute serially to other siblings, even if they execute concurrently.

A subtransaction can fail without causing the entire transaction family to fail. When a subtransaction fails, its parent can perform alternative processing, so the top-level transaction can ultimately commit.

Nested transactions have limited support in the industry today. Transarc's ENCINA offers an implementation of a nested transaction manager and a file system (Structured File System - SFS) which supports the nested model. Most of the object database systems have their own form of nested transactions but, to date, they do not operate with an independent transaction manager. The major database systems (Oracle, Sybase, DB2, Informix) can treat each data manipulation language (DML) statement as a subtransaction, but they do not currently support the generality of the nested model even when used with ENCINA as the transaction manager.

Nested transactions align nicely with the partitioning of application function provided by encapsulation within the object model. Each object has its own data which can only be changed by using the object's methods. If the data are to be subject to the transaction semantic, state changes will be deferred until the application commits, an operation which can also affect other objects. By using nested transactions, the object designer can provide transactional behavior for his object without worrying about other objects used by the application.

2.2 The OMG's OTS

A specification [3] for using transactions with distributed object technology was adopted by the OMG in December 1994. The specification, which was produced by collaboration between *Groupe Bull*, *IBM*, *ICL*, *Iona Technologies*, *Novell*, *SunSoft*, *Tandem Computers*, *Tivoli Systems*, and *Transarc Corporation*, defines an object service for creating and manipulating transactional objects.

2.2.1 Overview of OTS

OTS defines the interfaces for creating and manipulating transactions in an object environment. The OTS architecture is depicted in Fig. 4 and described briefly below.[2]

[1] For a more complete treatment, see [6].

[2] For a complete description of OTS, see [19].

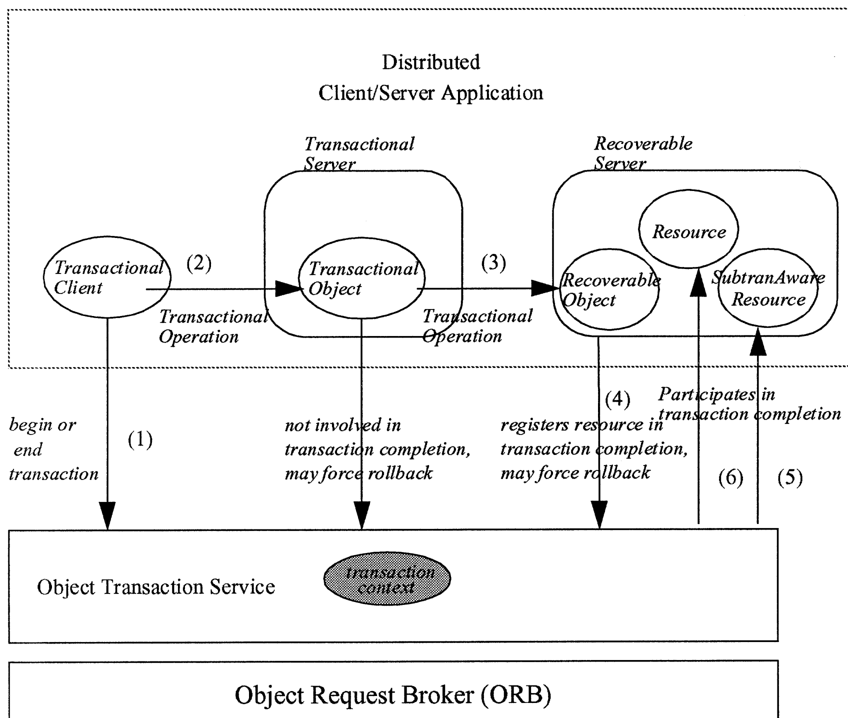

Fig. 4. OTS overview

A *transactional client* makes requests (1) of the OTS to define the boundaries of a transaction. It then invokes *transactional operations* (2) on *transactional objects*. A transactional object is an object whose behavior is influenced by participating in a transaction (e.g. an object that acts as a proxy for database activity).

During the life of a transaction, requests can be made (3) on *recoverable objects*. Recoverable objects are transactional objects which have recoverable state, i.e., state which changes atomically at transaction boundaries (e.g., an object which represents a customer account). Transactional objects are used to implement two types of application servers – transactional and recoverable.

- A *transactional server* is a collection of objects whose behavior is affected by the transaction, but has no recoverable state of its own.
  Transactional servers implement recoverable changes using other objects. A transactional server does not participate in the completion of the transaction, but it can force the transaction to be rolled back. Transactional servers may also start subtransactions.
- A *recoverable server* is a collection of objects, at least one of which has recoverable state.
  Recoverable servers implement transaction isolation by serializing access to recoverable state. A recoverable server participates in transaction completion by registering (4) a *Resource* object with OTS.

The *SubtranAwareResource* object implements participation (5) in the completion of a subtransaction. It is responsible for implementing actions which depend on subtransaction completion (e.g., assigning locks to the parent transaction).

The *Resource* object implements *transaction completion* by participating (6) in the two-phase commit protocol. It is responsible for durably recording changes to the state of its recoverable object and being able to recover those changes in the event of failure.

For each active transaction, OTS maintains a *transaction context* which it associates with transactional operations. The transaction context can be thought of as the control block which holds information about the transaction associated with a particular thread. It includes an identifier for the transaction, the state of the transaction (e.g. in flight, in doubt, etc.) and a list of resources which will participate in the commitment process. In OTS, the transaction context is represented by the *Control* object which also contains references to the *Coordinator* object (which controls registration and orchestrates the two-phase commit process) and the *Terminator* object which directs transaction completion.

Once a transaction is started, the transaction context becomes an *implicit parameter* of transactional operations and is transferred between the client and the transactional object by the ORB. OTS also permits the transaction context to be an *explicit parameter* of an operation.

An important characteristic of OTS is the ability for a single transaction to be shared between object invocations of OTS and procedural invocations, such as those defined by the X/Open DTP model [26]. This capability can be exploited to incrementally introduce objects into today's transaction processing environments.

2.3 Microsoft's transactions for OLE

Microsoft's OLE/TP [13] is another example of an object interface for transactions. Based on Microsoft's OLE (Object Linking and Embedding) architecture, it provides a series of C++ language interfaces to Microsoft's *Distributed Transaction Coordinator (DTC)*. DTC is being introduced with

SQL Server 6.5, but is intended to be incorporated in the Windows/NT and Windows/95 operating systems.

Based on a different object model (Microsoft's Common Object Model - COM), OLE/TP serves the same purpose in the Microsoft object architecture that OTS does with CORBA, viz. it provides a set of object interfaces for a transaction manager. The transaction manager (DTC) makes it possible for a variety of resource managers (i.e., those that support Microsoft's DTC interfaces for resource managers) to participate in a common transaction.

When distribution is provided in OLE (network OLE), OLE/TP will support a common transaction with different components on all Windows platforms. Inter-operability with CORBA could come at the network level when the OMG adopts a specification for COM/CORBA inter-operability.[3]

2.4 Transactional objects are important

At first glance, both OTS and OLE/TP appear to be object wrappers around the current procedural standards from X/Open. While true that OTS is based on the X/Open model [26], there are several important differences.

- Simplifications have been made to exploit the object environment,
  The X/Open standards carry a lot of "baggage" from existing implementations which do not fit naturally into the object environment. Since OTS has no legacy to carry forward, these were simply eliminated. OLE/TP appears to have taken a similar approach. Presumed nothing, chained transactions, and static registration have no counterparts in either OTS or OLE/TP.[4]
- Enhancements have been made specifically for the object environment,
  The object model introduces a powerful set of capabilities beyond procedural programming for building applications. To properly leverage these in a TP environment, OTS added nested transactions, explicit context management, and a recovery protocol driven by the resource managers[5] rather than the transaction manager. OLE/TP added explicit propagation and adopted a similar recovery philosophy.
- Inter-operability with procedural code was paramount for the OTS design,
  OTS is designed to permit a single transaction to include both recoverable objects and procedural resources (e.g., relational database tables) in the same transaction. To accomplish this, the transaction control structures used by existing transaction managers and resource managers must be mapped to their OTS equivalents and the identifier which uniquely distinguishes a specific transaction must be shared (or mapped) between the object domain and the procedural domain. The OTS architecture accomplishes this by allowing a common transaction manager component to support both objects and procedural resource managers. This enables existing procedural code to be *"wrappered"* and used *as is* in building new applications.

[3] OMG issued a Request for Proposal (RFP) for inter-operability between COM and COBRA in early 1995 and is expected to adopt technology in 1996.

[4] For more information on these concepts, see [6].

[5] Because the transaction manager is assumed not to have a log in the X/Open model, it must query each resource manager at restart to determine which transactions are in flight. In the OMG model, both the transaction manager and the resource managers are assumed to be logging the state data need for transaction recovery.

2.5 The value of nested transactions for objects

Once a flat transaction is started, all recoverable state updated by that transaction will be committed when the transaction commits. When objects are used to implement the transaction, the commit decision needs to be made externally to recoverable objects involved in the transaction, since commitment of the transaction affects the state of all objects, even if atomicity and durability are implemented by the database manager and not the objects themselves.

Listing 1 shows a simple implementation of a *Savings* object with recoverable state. *Savings* implements two methods, *query* and *debit* and has *account_num* as an attribute. The recoverable state of *Savings* is maintained by the *Database* object which acts as a proxy for a database manager which handles transaction recovery.

Listing 1. An object with recoverable state

```
typedef Amount float;

Amount query (){
 return ( database.read ( this->_account_num ) );
  }

void debit ( Amount toBeDebited ){
 Amount balance;
  balance = database.read ( this->_account_num );
  balance -= amount;
  database.write( this->_account_num , balance );
  return;
  }
```

Since the database requires a transaction to be active, the user of *Savings* must start a transaction before invoking its methods. When objects have their own recoverable state (i.e., state which is not persistently managed by the database), they must register to participate in transaction completion. This requires the object's implementer to specify how his object will be used and constrains the re-usability properties of objects.

- If invoked within a transaction, *commit* will affect its state and the state of all objects previously visited during the transaction.
  This requires the object implementer to depend on the commit decision being made external to his implementation, even if the object has recoverable state. If the user of the object fails to do so, the object's function is (at best) not performed or (worse) performed improperly.
- If invoked outside a transaction and the object has recoverable state, it must start a new transaction to cover its state changes.
  This requires the object implementer to include complex logic to cover its invocation outside of a transaction, and that assumes there is an easy way to find out whether a transaction is active or not.

Nested transactions provide a simpler alternative for implementing these kinds of objects, which increases their potential for re-use. The object implementer can bracket all its computations within a subtransaction. In OTS, the *begin* operation of the *Current* interface starts a nested subtransaction if a transaction currently exists, or a new top-level transaction if one does not. This allows the object implementer to delimit changes to its recoverable state and commit those changes, contingent upon completion of its ancestor transactions, if they exist, or immediately if they do not.

This is reflected in Listing 2, an alternative implementation of the *Savings* object in Listing 1. This alternative design effects only the debit method, so this code fragment does not show the query method which would not change.

Listing 2. An object with recoverable state using nested transactions

```
void debit ( Amount anAmount ){
  AccountNum theAccount = this->_account_num;
  Database database;
  Current theCurrentTransaction;

  theCurrentTransaction.begin();
  Amount balance = database.read( theAccount );
  balance -= anAmount;
  database.write( theAccount, balance );
  theCurrentTransaction.commit();

  return;
  }
```

Since the Listing 2 implementation of *Savings* affects its state and only its state, it provides a more rigorous implementation of encapsulation. This property of the OTS enables fine-grain objects to be created for each unique set of recoverable state and assembled in a variety of different ways to build business applications. Objects designed this way can be more efficiently re-used in constructing transactional applications.

2.6 Summary

Transactional object technology provides the infrastructure to develop transactional applications using objects. Both OTS and OLE/TP enable new transactional applications across a broad spectrum of computing platforms. The OTS design additionally facilitates evolutionary deployment in today's TP systems.

- OTS has wide support within the industry and can be expected to be available in products from multiple vendors[6] in the transaction processing marketplace on a wide variety of platforms.
- Looking at their success in deploying OLE to date, it can be expected that Microsoft will be equally aggressive in rolling out OLE/TP and products that depend on their transaction manager (DTC) on Windows platforms.

The definition of transactional object technology is a major step towards the pervasive deployment of TP applications built using objects. As customers and software vendors alike begin to adopt this technology, the traditional TP vendors need to be sure that their platforms can adopt to take maximum advantage of it. Failing to do so will severely restrict their future growth and may make them irrelevant in the object TP marketplace of the future.

[6] In fact, early products have begun to appear with implementations from Bull, IBM, Iona, and ICL available in some form as of early 1996.

3 Transactional workflow and objects

In commercial TP systems, a successful new technology requires both incremental adoption and the ability to integrate well with existing applications. The mission critical nature of these systems cannot accept more disruptive alternatives – no matter how compelling the new technology may be. Object technology has those attributes: it encourages incremental adoption through encapsulation and integrates well with legacy through a technique known as *"wrappering"*.

Today's re-engineering activities are focused on the definition of business processes. As these processes are identified and refined, two important characteristics emerge.

- Business processes are composed of multiple activities, tend to be long running, and involve people as well as computers. This leads to a requirement for new technology to define and automate the processes including the ability to track and restart individual activities in the event of failure.
- Many activities which comprised these business processes have already been automated. This requires a technology for encapsulating existing logic, so it can be made part of the new business process.

The most promising answer to the first is workflow, while objects are ideally suited to addressing the second.

3.1 What's wrong with transactions?

The traditional transaction models have several shortcomings which must be addressed to adequately support business processes as long-running transactions.

- Recovery protocols deal with the *recovery of data* and not the programs which manipulate the data. When a business process consisting of many activities fails before completion, *process recovery* is required to be able to re-start at a prior point of consistency.
- Both the flat and nested models assume transaction duration is short. This permits isolation to be achieved by *locking*. Locking prevents multiple transactions from changing the same data simultaneously by forcing contending transactions to serialize before accessing that data. Since update locks are held until transactions terminate, this strategy is most effective when transaction duration is short.

3.2 Advanced transaction models

But what if transaction duration is not short? Consider the example of a manufacturing production line. A product passes through many stages of assembly lasting for days

and perhaps weeks at a time before emerging as a finished product. Enabling these long-running transactions requires a different type of transaction. In the literature, these transactions are described as *advanced transaction models* [15]. These transaction models are formed by relaxing one or more of the traditional transaction properties.

3.2.1 Sagas

One of the first of these models to be described in the literature was the *saga* [5]. A saga is a sequence of ACID transactions, $\{T1,\ T2,\ \ldots\ Tn\}$. For each transaction in the sequence, the application writer defines a compensating transaction Ci which logically undoes the updates associated with Ti. A saga is rolled back by executing the compensating transaction of a committed transaction. When a failure occurs during the execution of a saga, the saga can continue after the failing transaction is recovered by re-executing the failing transaction and all subsequent transactions in the sequence.

Sagas relax the isolation property of ACID transactions, so locking will not prevent data from being seen by concurrently executing sagas. Locking occurs only for the individual transactions associated with each saga. Hence, data from any committed transaction can be read and updated by any other transactions. This means that the compensating transactions need to undo results logically, not physically, e.g., a credit of $100 to a bank account can be logically undone by posting a debit of $100, regardless of whether intervening credits or debits have been posted, although the posting of such debits could make the account balance negative, when it otherwise might not have done so.

But this can be handled. In fact, many banking systems work this way today, but the logic that tracks the need for compensation as well as the compensation itself is the responsibility of the application developer. Modelling these applications as sagas would not alter this design approach, since the mechanism for tracking and recovering process execution was never fully specified. Workflow systems address this issue by tracking the execution of a series of activities outside of the application logic and thereby relieve this burden from the application developer.

3.2.2 ConTracts

A *ConTract* [25] is a two-level transaction structure in which the top-level transaction is a persistent computation that initiates and coordinates flat transactions [6]. The control flow description (script), which specifies the sequence in which the transactions are to be executed, is a Pascal-like language which allows loops, conditional branching, and parallel actions. Though similar to the saga, it goes one step further by specifying failure processing and recovery.

The ConTract manager performs execution control at runtime using an event-oriented flow management technique based on a predicate transition net to specify activation and termination conditions for each step [6]. An important characteristic of the ConTract is that control flow and step coding are separated. This concept is embodied in today's workflow manager products and is key to the re-use of existing transactions in new business processes.

Although some of the concepts of sagas and ConTracts have been implemented in commercial products (e.g., most of the object database managers [17] have a scheme for long-running transactions based on check-out/check-in protocols and versioning), neither sagas nor ConTracts are available today in commercial products. Business processes are often long-running activities, so it is clear these concepts can be applied to business process re-engineering. *Workflow computing* [7] is emerging as the likely framework to incorporate these concepts. If so, workflow computing will be a key component of future transaction processing systems [12].

3.2.3 Implementing advanced transactions models with objects

Neither OTS nor OLE/TP include support for advanced transaction models in their current forms, but either might be used as an underpinning for building such facilities in the future. The key features such advanced transaction models should offer are

- an alternative approach to isolation
  Both OTS and OLE/TP rely on their resource managers to provide isolation. *Object Concurrency Service* (OCS) is the OMG object service which provides the required function. OCS supports classical two-phase locking protocol [6]. OLE/TP does not specify how locking is accomplished. Novel types of synchronization are required for long-running transactions (e.g. explicit handling of conflicts when they occur).
- a mechanism for process recovery as well as data recovery
  A process manager that provides such facilities can be implemented using the features of a transaction manager to durably protect the process definition and current state, providing that definition can be captured externally to the application.

3.2.4 Workflow computing

Workflow computing [7] is an emerging technology for implementing long-running business processes. Although it has its origins in office automation and deals with people as well as information systems, it can be used as a discipline for connecting multiple applications, including existing short-duration transactions, according to a pre-defined flow of control.

A workflow can be thought of as a *long-running transaction*, but, more accurately it is a series of (independent) traditional transactions whose execution proceeds in some predictable sequence with a well-formulated set of interdependencies between them. Complex applications involve many smaller application systems which are closely related to each other. Managing the flow of work [11] through these interrelated activities can improve organizational productivity, reduce costs, and guarantee accountability. It also allows existing systems to be integrated, ensuring that the business processes they represent are, in fact, connected to each other.

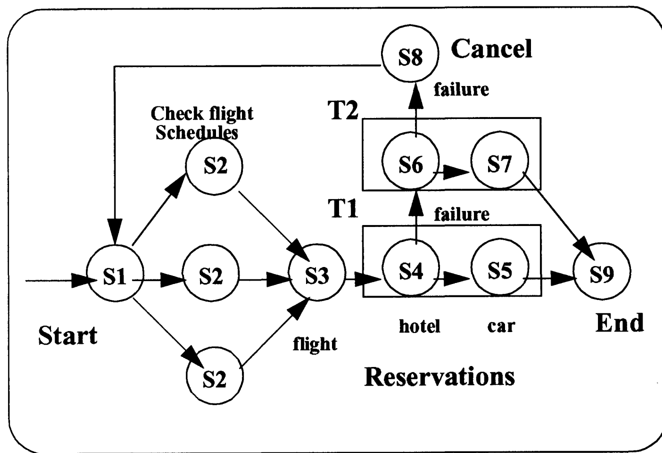

Fig. 5. Booking a business trip

3.2.5 The problem

The short-duration assumption inherent in the traditional transaction models does not fit well with a large class of applications. Consider the simple example in Fig. 5 of booking a business trip (step S1). An airline reservation (S3), a hotel (S4 or S6), and a rental car (S5 or S7) are all required (S9), and need to be lined up with each other or the trip will not be taken (S8). Since the inventory for each of these items is maintained by its respective owner, a single flat or nested transaction cannot reasonably cover all three sets of changes. The application (in most cases a person in the role of a travel agent), must implement the correct sequence of traditional transactions to ensure a reasonable outcome, viz.

- a seat is reserved on a flight to your intended destination,
- your car is available at your destination airport, and
- your hotel room is reserved for the evening of your arrival at your intended destination.

Failures require that the *process be recovered*, not just the data. A plane reservation without the waiting rental car or the evening hotel room is not good enough.

Workflow computing addresses this class of problem by providing:

- a mechanism for specifying the relationship between the steps, which is external to the implementation of the steps themselves, and
- a *process manager* which durably records the execution of the various steps and continues the sequence in the event of failure.

3.2.6 Business processes are disjoint

The example in Fig. 5 demonstrates an important characteristic of business processes. **The steps are disjoint, not only in time, but also in their ability to share data.** The reason this example cannot be implemented by a flat or nested transaction in the real world is not technical. One can imagine a number of different design approaches that will guarantee the plane, car, and hotel are synchronized using traditional transaction technology. But they all have the property that inventory data owned by multiple enterprises (the airline, the rental car agency, and the hotel chain) will be locked by an external user (the travel agent), preventing that data from being used by anyone else, including its owner! It's not surprising that every business finds this unacceptable, even before someone explains the details of transaction recovery for failures in the in-doubt state.[7]

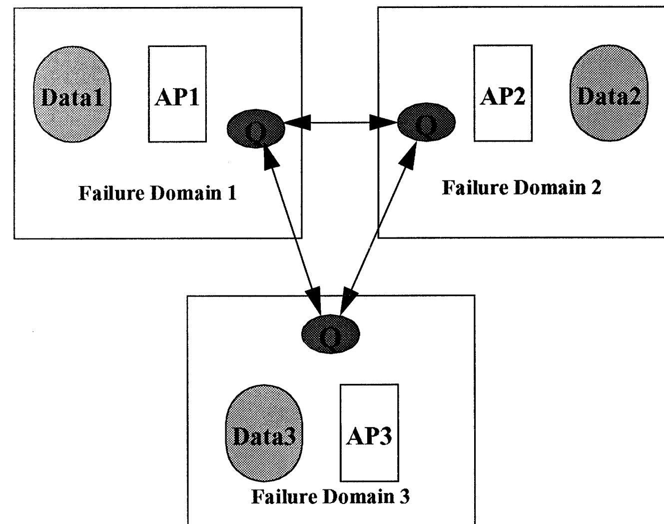

Fig. 6. Transactional messaging

Even within a single company, decentralizing decision-making requires individual departments to have control over their operational data while implementing well-defined procedures for sharing that information with others. This satisfies the corporation's need for timely information, while ensuring that each unit can successfully execute its business processes efficiently.

3.2.7 Connecting activities by messaging

The technological consequence of these business requirements is an **increased importance for messaging technology** as the principal mode of communication between business entities. *Transactional messaging* allows transactions to be localized to failure domains with independent recovery scopes. In other words, a single failure never requires coordinated recovery between independent business entities. Transactional messaging provides "exactly once" message delivery by including the message in the commit scope of the local transaction. This is typically accomplished by using a recoverable message queue as shown in Fig. 6.

However, this independent failure property comes at the expense of traditional transactional integrity for the complete process, requiring a higher level of transaction control external to the applications themselves. The higher level of control is implemented in an external *process definition*.

[7] The two-phase commit protocol requires that a resource manager wait for the decision from its coordinator once it has been prepared. Loss of connectivity can make the elapsed time for discovering the transaction outcome quite long, but data integrity cannot be guaranteed if the resource manager decides to unilaterally commit or roll back (take a heuristic decision). See [6] for more details.

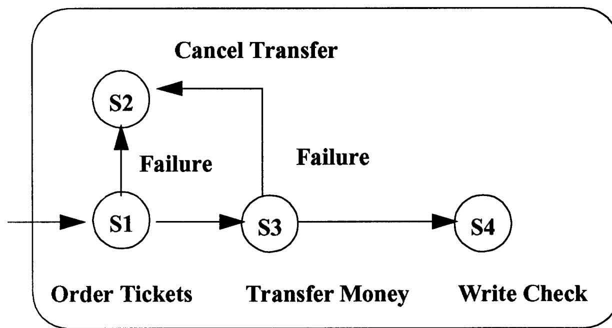

Fig. 7. Buying playoff tickets

3.3 Object wrappering

Each step in the business process in Fig. 5 can be implemented by a traditional short transaction. In fact, many already are. The ability to incorporate existing transactions "as is" is powerful leverage when building more complex business applications. A simple example of re-use by object wrappering is detailed in the following sections.

3.3.1 Using wrappers to build a business process

Your favorite sports team clinches a playoff position and announces that playoff tickets will go on sale tomorrow morning. You absolutely want to go, if you can obtain tickets, but you have insufficient funds in your checking account. In order to be able to buy the tickets, you will need to transfer money from your savings account. This requires you to:

1. call the ticket agency and reserve your tickets (if available),
2. transfer the money to your checking account, and
3. write the check for the tickets.

This process (which can be modeled as either a saga or a ConTract) consist of the above three steps and they must be completed before the game is played or the tickets have no value. This introduces the notion of *timeliness of execution* which is inherent for traditional (short-duration) transactions, but which needs to be dealt with explicitly when defining workflows.

You have implemented a short-duration transaction to transfer the money (S3), so you need to implement the other two steps (S1 and S4) and define the sequence to a process manager, so that all three will be executed. If it happens that you cannot get the desired tickets or game day arrives before the process completes, there is no reason for the savings funds to be in your checking account, so they will be returned to savings. Such an action is implemented by a different transaction (S2) and is termed a compensation. The sequence diagram which represents this process is shown in Fig. 7.

To utilize the transfer account function, we need to provide a wrapper for it. The wrapper encapsulates the *fromSavingsToChecking* activity and defines a *transfer* method which takes the transfer amount, *transfer*, as an input parameter. This is depicted in Listing 3.

Listing 3. Transfer funds to checking account

```
Boolean fromSavingsToChecking ( Amount transfer ){

 Current theCurrentTransaction;
 Savings theSavingsAccount;
 Checking theCheckingAccount;

 theCurrentTransaction.begin();
 Amount balance = theSavingsAccount.query();

 if ( balance > transfer )
  { theSavingsAccount.debit( transfer );
   theCheckingAccount.credit( transfer );
   theCurrentTransaction.commit();
   return True; }
 else
   theCurrentTransaction.rollback();
   return False; }
 }
```

3.3.2 Defining new transactions

Our example requires three addition transactions to implement the business process of buying our playoff tickets:

– S1 checks the availability of tickets and reserves them. This needs a *query* function against the ticket inventory and a *reserve* function to hold them.
– S2 undoes the transfer if tickets are not available. This is almost the same as the existing transfer function (S3), which moves funds from savings to checking, but instead transfers the funds from checking to savings.
– S4 orders the tickets previously reserved. This needs a *buy* function that operates on the tickets previously held by *reserve*.

In the interest of simplicity, only the client code of each transaction is shown. Because we need to include all of these transactions within our process definition, we give each of the transactions a wrapper, similar to the one for the *fromSavingsToChecking* transaction in Listing 3.

Ticket availability: the transaction to determine ticket availability is shown in Listing 4.

A *query* of the *Tickets* object determines if the required number of tickets are available. If so, a *reserve* for the requisite number of tickets is made and a reservation ID, *recid*, is returned. The *buyer* transaction will use this to *locate* the ticket reservation before *buy* is executed against the ticket inventory. If the necessary number of tickets are not available, the transaction is rolled back.

Reversing the transfer: the reversing transaction is a mirror image of the original transfer transaction (Listing 3). A *debit* operation is added to the *Checking* object and a *credit* operation is added to the *Savings* object.[8] This enables the client application to move funds from the checking account to the savings account.

Buying the Tickets: the buying transaction is shown in Listing 5

[8] One observes, of course, that *Savings* and *Checking* objects are merely specializations of the *BankAccount* objects, so the debit and query methods could be inherited rather than rewritten.

Listing 4. Ticket availability transaction

```
RecId available ( int required ){
 Current theCurrentTransaction;
 Tickets theTickets;
 RecId result;

 theCurrentTransaction.begin();
 if ( theTickets.query() > required )
  { result = theTickets.reserve( required );
   theCurrentTransaction.commit();
   return result; }
 else
  { theCurrentTransaction.rollback();
   return RecId::NullRecId; }
 }
```

Listing 5. Buy the tickets

```
Boolean buyer ( RecId theRecId ){
 Current theCurrentTransaction;
 Tickets theTickets;

 theCurrentTransaction.begin();

 if ( theTickets.locate(theRecId) )
  { theTickets.buy( theRecId );
   theCurrentTransaction.commit();
   return True; }
 else
  { theCurrentTransaction.rollback();
   return False; }
 }
```

The *buyer* transaction uses the *recid* returned by the availability transaction to verify the reservations and subsequently debits the ticket inventory to finalize the order.

3.3.3 Defining the process

Since each of the steps in our process was defined as individual transactions, the recoverable data associated with each of them will always be made consistent in the case of (individual) failures. In other words, if we succeed in making the reservation, the tickets object will have a record of our reservation. However, unless we also executed the buy, it will have no knowledge of the tickets being bought. Unless something remembers that the business process needs all three steps to be successful, our process will never complete.

The system component that implements this memory is a *workflow manager*. A workflow manager provides a set of tools which enable a set of activities (the steps), the relationships between them, and compensations in the event of errors to be defined external to the applications which implement the business logic. Hence the *fromSavingsToChecking* transaction need not concern itself with the need for compensation if the tickets cannot be obtained. In fact, it need have no knowledge at all of why the funds are to be transferred, allowing it to participate in other business processes as well.

An example of a workflow manager is IBM's FlowMark [8]. Flowmark consists of an interactive tool for defining the activities in a business process and their relationships as well as a workflow engine which executes the activities and tracks the progress of the process. This enables process recovery to be performed if any of the steps fail to complete properly. To date, the dependencies between these activities, which are, in fact, transactions, have not been embedded in the workflow engine. This requires the human operator to initiate compensation when it is required. Future workflow systems are likely to be enhanced to support this notion of *transactional workflows*.

The FlowMark model for the workflow consists of the activities *Order Tickets*, *Transfer Money*, *Cancel Transfer*, and *Write Check*. The activity *Order Tickets* receives as input the number of tickets to be ordered (*NumTickets*) and produces the Boolean *ordered* as output, while the activities *Transfer Money* and *Cancel Transfer* receive the amount to be transferred or canceled (*Amount*) as input.

Order Tickets and *Transfer Money* can be run in any order (specifically, *Transfer Money* could run before *Order Tickets* produces a positive result), which makes it necessary to compensate in case the order cannot be satisfied.

The compensation activity *Cancel Transfer* is run if and only if the condition *ordered=NO* is fulfilled when *Order Tickets* terminated (assume an appropriate exit condition of *Order Tickets* like *ordered=YES* or *ordered=NO*) and the *Transfer Money* activity[9] has run. For this purpose, the start condition of the activity is the conjunct of the transition condition of its incoming control connectors.[10]

In case the tickets can be ordered (i.e., the transition condition *ordered=YES* becomes true), the *Write Check* activity could be started, but the *Transfer Money* activity must have terminated before *Write Check* becomes startable.

3.4 Summary

The ability to take existing transaction implementations, whether implemented using object techniques or not, and create object wrappers for them, allows these transactions to be used as activities in larger business processes. Traditional transaction *data recovery* ensures that recoverable data is always made consistent after failure of any step.

A workflow manager allows these processes to be described externally and contains the functions necessary to perform *process recovery* in the event of failure. Typically, these processes will be defined by people from business administration, without regard to how (or if) they are represented in an information system. The proper recovery aspects will be added by people with different skills. The resulting process contains the proper combination of data and process recovery to make it possible to transform complex business processes into information systems representations.

Workflow computing, with transactional messaging and the use of transactional object technology enables the construction of these commercial TP applications. Together, they permit the enterprise to rapidly develop information systems based on the enterprise business model, which are adaptive to changes in the business environment.

[9] No explicit specification of a transition condition results in a constant "TRUE" predicate.

[10] Start conditions are not explicitly shown in Fig. 8.

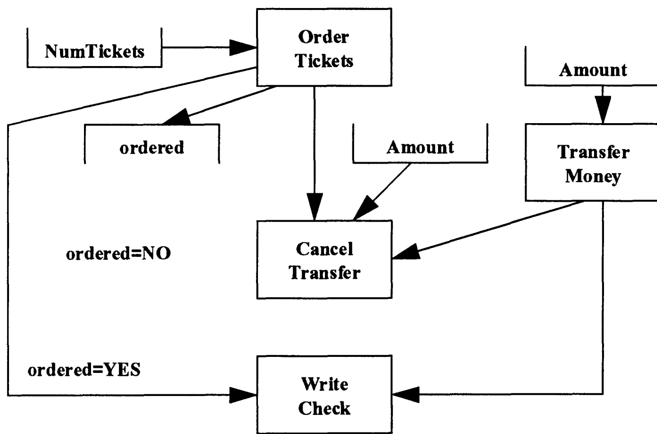

Fig. 8. Buying playoff tickets as a FlowMark workflow

4 TP monitors and ORBs

TP monitors[11] provide an environment for executing many of the commercial applications which utilize transaction technology. A TP monitor can be characterized as a lightweight scheduler, one that is optimized for the short-running applications favored by the flat and nested transaction models. TP monitors perform a *"traffic cop"* function [1] which allows a large number of clients to efficiently access a much smaller number of application servers [6].

4.1 Evolution of TP monitors

TP monitors were first deployed on centralized mainframes as a way of supporting large numbers of terminals. IBM's Airline Control Program (ACP), introduced in 1964, was the first TP monitor. ACP supported online airline reservations and became the benchmark for high-volume transaction processing for several decades. Today, as the Transaction Processing Facility (TPF), it still supports many of the world's airlines.

4.1.1 The age of centralized computing

In the centralized systems of the 1970s, processor cycles and storage were critical resources. Efficiently managing user access to these resources was the problem to be solved and the TP monitor solved it by allocating these critical resources dynamically in response to each terminal input. Each terminal user appeared to have immediate access to any program to which it was authorized. Scheduling users to processes (a *processcentric* system) enabled critical systems resources to be efficiently shared by a large number of terminal users. Data was owned by the TP monitor (it OPENed the files) and was shared among the terminal users by locking.

[11] Most TP monitors are packaged in transaction processing systems. Such systems will contain the TP monitor, a transaction manager and systems management facilities. Often a user interface package will also be present, especially when non-programmable terminals are supported. Resource managers such as queuing systems or file systems may also be present. IBM's CICS, for example, contains all of the above components as well as a unifying API which mediates access to each of them.

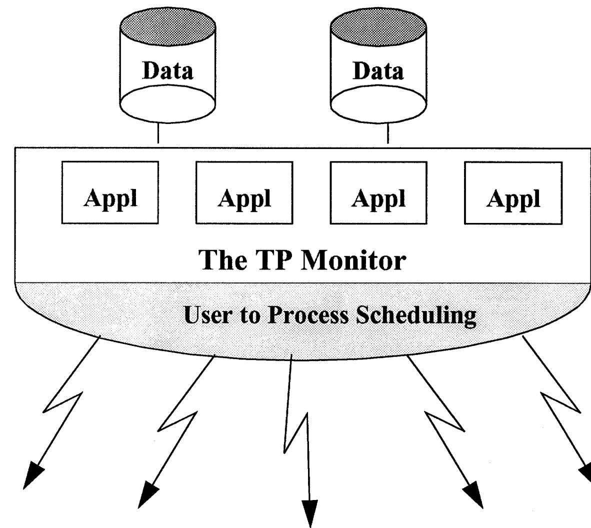

Fig. 9. A centralized system

Many commercial applications were implemented on these TP monitors. Products like IBM's CICS and IMS, DEC's ACMS and Intact, and Tandem's Pathway were largely responsible for the rapid growth of mainframes and later mini-computers during the 1970s and 1980s. In fact, IBM's CICS and IMS became so pervasive in the MVS environment that today almost every MVS system has at least one, and many have both.

Application development was relatively simple. The TP monitor allowed each application to be written for a single user, ideal for languages like COBOL which do not natively support multi-threading or non-blocking I/O. All application code executed in a single location. The TP monitor provided a simple view of the complicated part of the system, viz. terminal communications, and handled the difficult task of making sure all the terminals were serviced. The user could concentrate on the business problem to be solved.

4.1.2 The dawn of distributed computing

With the advent of the personal computer, each terminal user had his own processor cycles, memory, and storage as well as the requisite programs. Cycle sharing was not a problem. Sharing disks (or more specifically data) and printers attached to *local area networks* was the new challenge. The emphasis shifted to scheduling program access to data (a *datacentric* view). The data providers (both the file systems and the database vendors) became dominant, since they provided the scheduling functions users needed.

Application development was again simple. An application still supported a single user, although high-function graphical interfaces demanded multi-threading and non-blocking I/O, which is more difficult in COBOL. The application still executed in a single location. The database products provided a simple view of the real complexity – access to shared data – via simple programming interfaces. The user could concentrate on the business problem to be solved. The application had shifted from the machine room to the desktop!

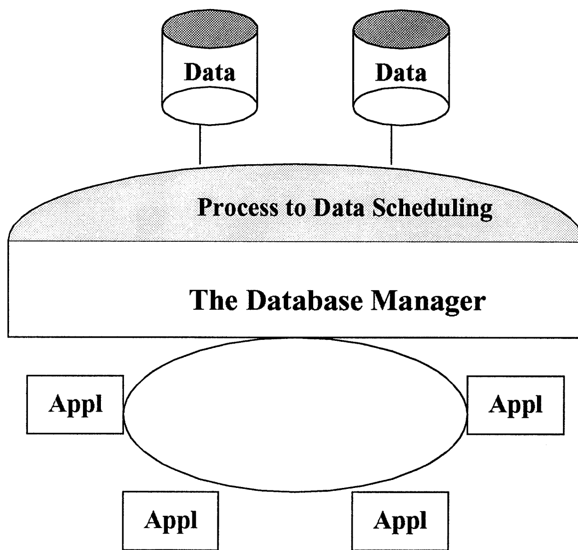

Fig. 10. A client/server system

4.1.3 Distributed function, a better (but harder) solution

By the end of the 1980s, the emphasis had shifted again. The departmental server could execute application function as well as the desktop, and it was more efficient to have it do so. The departmental systems also needed access to other systems both within and outside the enterprise. Efficiency was gained by packaging multiple requests for data and sending only a single request, rather than one per data access, in other words, executing application logic in multiple places or *distributed function*.

Distributed function is an appealing systems architecture. It maximizes the value of each communication message by executing application logic closest to the data it operates on.

TP monitor approach to distributed function. Two different distributed function architectures began to emerge. From the processcentric world of the TP monitors, new programming interfaces for *interprocess communication* were delivered.

- IBM's Advanced Program-to-Program Communications (APPC) was one of the first.
  The APPC model is built on the familiar *conversation* paradigm (like a telephone call). A connection is established, the parties take turns speaking, and the connection is terminated. APPC provides a powerful set of application functions [9] to enable multiple applications to communicate and optionally share a transaction across multiple sites connected by SNA communications. The same concepts were subsequently adopted and embellished as part of the ISO OSI-TP standard [10].
- Remote Procedure Call (RPC) became very popular.
  RPC extended the familiar *language call* paradigm to allow subroutine execution in a different machine than the calling program. Initially popular in the UNIX world, RPC became the cornerstone of the Open Software Foundation's (OSF) Distributed Computing Environment [20] (DCE) and has been subsequently endorsed by major IT vendors, not only for their UNIX platforms, but for their PC and mainframe systems as well.
- Messaging enjoyed a rebirth[12] in popularity.
  Messaging is a style of program-to-program communication which does not depend on the real-time availability of a communication path. Application programs are represented by *queues* which are implemented on durable media. The sending application places a message on a queue and (later) reads its reply from a different queue. The receiving application reads its input queue (the one written by the sending application), processes the message, and generates a reply. Messaging is like electronic mail, which delivers information relatively quickly when communications facilities are available, and ultimately when they are not. Messaging provides *guaranteed delivery* among independent transactions rather than requiring a single transaction to be shared between communicating programs.

Much can be and has been written about the merits of each of these approaches [21], but, to a greater or lesser degree, they all have the characteristic that application development is difficult.

- Conversation and to a lesser degree RPC require function placement decisions to be made early and, once made, those decisions are not easily changed.[13]
- With conversation, the communications programming which ties the pieces together is complex and requires expensive skills.
- To date, only RPC has provided data conversion mechanisms to insulate applications from the different representations of various types of data (characters, integers, floating point, etc.).
- Application development tools support RPC but are generally lacking for conversation and messaging.

The database approach to distributed function. From the data centric world of the database vendors came a novel approach – put the programs in the database! *Stored procedures* are extensions to the SQL language that allow varying degrees of procedural logic to be executed as part of an SQL call. A functional extension of database *triggers*, which are used for expressing integrity constraints and business rules (e.g., "customers that owe us money cannot be deleted from the customer database"), stored procedures can be used to execute business logic within the database server. Some will argue [1] they should be used for developing all application function. Their objective is the same as the TP monitor solutions – to minimize the number of communication messages and make multiple data accesses as a result of a single request.

Stored procedures have a lot in common with RPCs. Data access is a subroutine call and stored procedures are built on that paradigm to do more database work in a single statement, while maintaining the simplicity of SQL. Stored procedures have some disadvantages though.

[12] It's worth noting that messaging systems have existed for some time, although most implementations were unique to a single vendor and a single platform. IBM's IMS, for example, has had a messaging architecture for communications since its introduction in 1968. Tandem has had messaging hardware in support of Pathway since its introduction in the mid-1970s.

[13] Administrative facilities may be available to alter where a specific piece of logic is to be executed, but they cannot compensate for improper function decomposition bound in at design time.

– Stored procedure languages are proprietary to a specific database vendor. Stored procedures were developed by each of the database vendors to specifically exploit their products. Once a decision to use one vendor's offering is made, the ability to change to a different vendor's database product is difficult or impossible. Stored procedures will be part of the SQL3 standard, providing for more portability between database manager implementations.
– SQL as a programming language is limited. Unlike COBOL or C, SQL is not a complete programming language[14], so programmers are limited in what they can do. Arithmetic operations are lacking, and, although many vendors offered the ability to code in some programming language (typically C or BASIC), none offer the choice of application development languages supported by the TP monitors.
– Coordinated access to recoverable resources other than the vendors database is not possible. Database products have not elected to provide the functions of an external transaction manager for resources other than their own databases. As a result, a coordinated two-phase commit involving more than the data in a single vendor's database cannot be accomplished, even on a single machine.[15]

Even with all of the above, stored procedures suffer from many of the same problems as the inter-process communications interfaces of the TP monitors.

– Function placement decisions are still made early and are not easily changed.
– The programming which ties the pieces together may be complex and requires extensive knowledge of the database vendor's stored procedure implementation.

However, application development tools are provided by the database vendor, so application development is not quite as difficult.

4.2 The object request broker

ORBs provide another mechanism for clients to communicate with servers. An ORB hides details of communications from its users. In the OMG's OMA, the ORB[16] provides both a static and dynamic form of invoking operations on objects.

– The Dynamic Invocation Interface (DII) supports application requests constructed at runtime and then passes them to the ORB. This enables the business application to defer to runtime the identification of the objects it will interact with.

[14] Although SQL3 will offer many functions which currently require a programming language.

[15] Modern operating systems such as Digital's VMS, IBM's OS/400 and MVS, and Microsoft's Windows/NT are offering external transaction managers as part of the operating system, making it likely that future stored procedures implementations will be able to coordinate multiple resources.

[16] For more information, see [18].

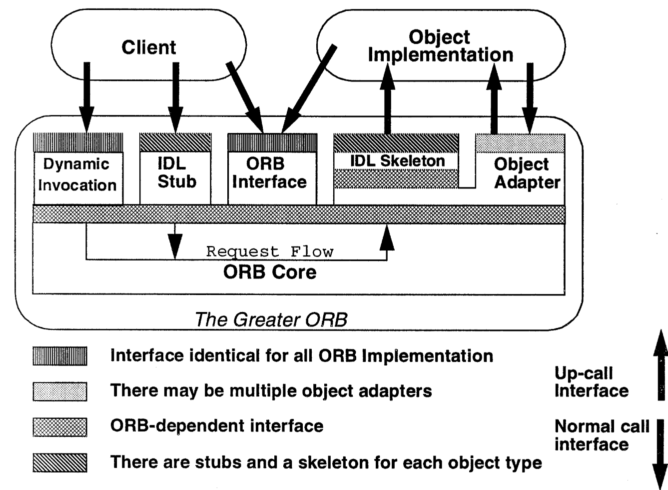

Fig. 11. Common object request broker architecture (CORBA)

– Interface Definition Language (IDL), a syntax similar to the C programming language, describes the interfaces provided by an object implementation – the operation's *signature* – external to its source language implementation. IDL stubs are built by the IDL compiler at development time and bound into the client's executable code.
– On the server side, IDL skeletons provide the linkages between the ORB and a particular object implementation. They are also built by the IDL compiler at development time and bound into the object's executable code.
– The object adaptor is responsible for implementing a particular object activation policy.
– On the client side, object invocation is similar to an RPC.[17]
– The client makes a local call to a stub (or proxy) which represents the target object.
– The target object is located using a naming service and a path to the object selected.
– The method parameter list is marshalled into a linear string defined by the method's signature and sent to the target system.
– The client thread is blocked until the response is returned.
– When the response is returned, the data is demarshalled into a parameter list defined by the method's signature.
– The client's call is then re-dispatched.

It is on the server side that the characteristics of a TP monitor begin to emerge, particularly in the CORBA function of the *object adaptor*.

4.2.1 Object adaptors

Object adapters provide for the activation of objects in different execution environments. CORBA defines different types

[17] CORBA today is defined in terms of operations and their results which can be either return values or exeption status. In language terms, this is a procedure call.

of object adapters [18] to "provide flexibility in how an object is activated." An object adapter, "is the primary interface that an implementation uses to access ORB function". CORBA defines the Basic Object Adaptor (BOA) interface, "that is intended to support a wide variety of common object implementations". CORBA further defines[18] that, "more (but not many) object adaptors will be needed to support different kinds of object granularities, policies and implementation styles".

An object adapter provides flexibility in how an object is activated, e.g.,

- the creation of a new process,
- the creation of a new thread within an existing process,
- the re-use of an existing process or thread.
- or even a switching function that allows a large number of object clients to be supported by a smaller number of object servers (sound familiar?).

Each of these forms of activation have differing performance, availability, and manageability characteristics.

4.2.2 Object invocation

In many respects object invocation is like the inter-process communication functions of the TP monitors.

- Methods are invoked as the only way of accessing data. Data access occurs as a by-product of process execution. In contrast, the user of a database system identifies the data and then the operation which acts on it.
- Multiple clients may access a single object. This requires a mechanism for efficiently mediating access, just like the user to process scheduling functions of a TP monitor.

4.2.3 Object activation

In order to activate an object, an environment to contain the object must exist. It might exist because it has been started in advance (like a server program), or it can be made to exist at the time the activation is requested. Both possibilities are allowed for.

If we separate the functions of *process creation* (or scheduling) from the functions of *object activation*, we begin to see how a marriage between the two technologies might come about. Object activation is a unique function of the ORB environment, while scheduling is more generally applicable. TP monitors provide scheduling function today with a myriad of performance, availability, and manageability characteristics. TP monitor technology can be used tomorrow to make these characteristics available to ORB environments as well.

4.2.4 Comparing object servers to TP monitors

Today's generation of ORBs rely on a server model similar to today's non-TP monitor environments.

- A finite number of server processes are started by some external mechanism (e.g., an operator command).
- A "listener" thread within the ORB monitors the communication transport for incoming requests and routes them to the correct server process.
- Each server process must be able to handle multiple users so it will be forced to engage in some form of multithreading within its implementation.

This approach has several disadvantages.

- Application programming is more complex.
  Each object implementation must cater to its simultaneous use by multiple users. This requires multi-threading, concurrency control (to access shared resources), and application level security. In other words the application starts to include many of the functions of today's TP monitors.[19]
- Scalability is limited.
  As new applications are added, new server processes are required. New clients can also require additional servers just to handle the increased workload. Since the client to server bindings are static, one quickly runs out of the ability to add additional servers, either through diminished processing power, or memory, or both.

4.3 TP monitors

A TP monitor provides many of the functions envisioned for the object adapter in CORBA. The scheduling function provided by TP monitors include:

- assignment of client requests to a particular application server. This can include process creation as well as reuse of existing processes
- initialization of the program to handle the client request. This can include program loading and dispatching

These scheduling functions allow TP monitors to improve system performance not only in large systems with tens of thousands of clients but in smaller systems as well. A variety of scheduling algorithms have been implemented by commercial TP monitors. Each algorithm delivers different value and supports different customer requirements. Some examples:

- using separate processes for application servers allows the hardware to provide isolation of individual applications and thereby deliver greater reliability and availability
- using different instances for each user allows the application to be implemented for a single user and can avoid multi-threading
- selecting processes from a pre-allocated pool rather than creating them at the time of the client request improves performance

[18] Cf. Chapter 9 of [18].

[19] SAP R/3 is an example of how these TP monitor functions have migrated into the application program even without the use of an ORB.

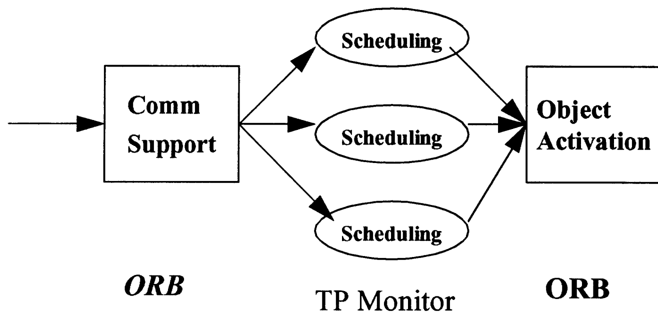

Fig. 12. Integrating TP monitors and ORBs

4.4 Combining ORBs with TP monitors

One way to combine the functions of ORBs with TP monitors is to separate the scheduling functions of the object adaptor from the activation functions and encapsulate the TP monitor scheduling technology in the object adaptor. This is depicted in Fig. 12.

There are potentially a very large number of scheduling algorithms. Separating these algorithms from the type of work being scheduled allows them to be incorporated as required to meet the needs of different application environments. When applied to the object environment, it yields the following advantages:

- efficient scheduling of the environment in which objects are to be activated. This is essential if objects are to be used to build traditional OLTP applications
- the ability to define and use objects independent of the TP monitor selected. An essential element for a component software market in the OLTP environment
- the ability to enhance the basic object environment or the TP monitor without impacting the functions of the other
- the ability to integrate two compatible technologies in a way that best addresses the needs of the commercial TP environment

4.5 Summary

TP monitors and ORBs are indeed compatible. In fact, the two technologies complement each other quite nicely. And, because of the tool sets associated with creating and deploying objects, application development is easier than any of the current TP monitor inter-process communications techniques.

Just like the classical mainframe environment where the TP monitor provided the ability to efficiently build and execute commercial applications, the marriage of ORB and TP monitor technology – *the object TP monitor* – can create a new distributed computing infrastructure, exploiting the traditional strength of the TP monitor, while providing a robust application development environment based on re-usable objects. The marriage of the two can produce an integrated solution that optimally addresses the needs of the commercial TP environment.

5 Conclusions

The marriage of transaction and object technologies will have a profound impact on commercial TP. Synergy between the business model and its object representation in a supporting IT solution holds great potential since it

- can be more closely tuned to the needs of the enterprise
- is not constrained by the architecture of the computer system chosen for implementation
- is easily adaptable to changes in the business environment

Applying object technology to the construction of commercial TP applications will enable the enterprise to realize the benefits of distributed computing as well as construct these applications more rapidly.

The ability to model complex business processes as workflow and to integrate legacy applications into new business applications permits incremental adoption of the technology and delivers maximum leverage of the existing IT investments. This technology base will enable the next generation of transactional applications to permit businesses to be more responsive to changing market requirements.

The object TP monitor will provide a robust environment for deploying these applications which capitalizes on the strengths of today's TP monitors while preserving the benefits inherent in object technology.

Acknowledgements. I would like to acknowledge the contributions of my colleagues at IBM; Cindy Saracco and Franz Spickhof of the Santa Teresa Laboratory, San Jose, California, Iain Houston, Keith Jones, and Susan Malaika of the Hursley Laboratory, Hursley, UK, C. Mohan of the Almaden Research Center, San Jose, California, and Frank Leymann of the German Software Development Laboratory, Böblingen, Germany.

References

1. Cameron B, Woodring S (1994) OLTP meets client/server, Volume 5, Number 5, The Software Strategy Report, Forrester Research, Inc.
2. Cobb E (1994) Introducing Transactions to Objects, Middleware Spectra
3. Cobb E, et al. (1994) Object Transaction Specification. OMG Document TC 94-8-4
4. Dixon G (1994) Object Concurrency Specification. OMG Document TC 94-5-8
5. Garcia-Molina H, Salem K (1987) Sagas. In: Dayal U, Traiger I (eds) Proceedings SIGMOD International Conference on Management of Data, San Francisco. ACM, NY, pp 249–259
6. Gray J, Reuter A (1993) Transaction processing: concepts and techniques. Morgan Kaufmann, San Mateo, Calif.
7. Hsu M (Ed.) (1993) Special Issue on Workflow and Extended Transaction Systems. Data Eng vol 16, no 2, IEEE Computer Society, Washington, DC, pp 3–56
8. IBM Corporation, FlowMark for OS/2: Modeling Workflow, SH19-8175
9. IBM Corporation, SNA Transaction Programmer's Reference Manual for LU Type 6.2, GC30-3084
10. International Standards Organization (1992) Open Systems Interconnect – Distributed Transaction Processing (OSI-TP). ISO IS 10026
11. Leymann F, Roller D (1994) Business Process Management with FlowMark. In: Werner R: Digest of Papers. IEEE Compcon, San Francisco, IEEE Computer Society Press, Los Alamitos, Cal., pp 230–234

12. Leymann F (1995) Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems. In: Lausen G (ed) Proceedings BTW95 Databases in Office, Engineering and Science. Springer, Berlin Heidelberg New York, pp 51–70
13. Microsoft Corporation (1996) Microsoft Distributed Transaction Coordinator Resource Manager Implementation Guide, Version 6.5
14. McClure S (1993) Object Technology: A Key Software Technology for the '90s. International Data Corporation White Paper
15. Mohan C (1994) Tutorial: advanced transaction models – survey and critique. In: Snodgrass R, Winstett M, Proceedings presented at ACM SIGMOD International Conference on Management of Data, Minneapolis, Mn., ACM Press, NY, p. 521. Http: //www.almaden.ibm.com/cs/exotica/exotica-papers.html
16. Moss E (1985) Nested transactions: an approach to reliable distributed computing. MIT Press, Cambridge, Mass.
17. Object Data Management Group (1993) The ODMG Standard
18. Object Management Group (1995) The Common Object Request Broker: architecture and specification. Object Management Group
19. Object Management Group (1996) CORBAservices. Object Management Group
20. Open Software Foundation (1990) OSF Distributed Computing Environment Rationale
21. Orfali R, Harkey D (1994) Client/Server Survival Guide for OS/2. Van Nostrand Reinhold, New York
22. Orfali R, Harkey D, Edwards J (1996) The Essential Distributed Objects Survival Guide, Wiley, New York
23. Tibbetts J, Bernstein B (1994) Toward a Standard for Transactional Objects: An Interview with IBM's Ed Cobb, Open Transaction Management Spectrum, pp 37–41
24. Taylor DA (1995) Business engineering with object technology, Wiley, New York
25. Waechter H, Reuter A (1992) The ConTract Model, In: Elmagarmid A (Ed) Database Transaction Models for Advanced Applications. Morgan Kaufmann, San Mateo, Calif., pp 217–263
26. X/Open Corporation (1992) Distributed Transaction Processing: Reference Model, X/Open Ltd.
27. X/Open Corporation (1994) Distributed Transaction Processing: TX Specification, X/Open, Ltd.
28. X/Open Corporation (1994) Distributed Transaction Processing: XA Specification, X/Open, Ltd.