Exotica: A Project on Advanced Transaction Management and Workflow Systems*

C. Mohan D. Agrawal† G. Alonso
A. El Abbadi† R. Günthör M. Kamath‡

IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA
{*mohan, agrawal, gustavoa, amr, rgunther, kamath*}*@almaden.ibm.com*

Abstract

This paper is an overview of the Exotica project, currently in progress at the IBM Almaden Research Center. The project aims at exploring several research areas from advanced transaction management concepts to client/server architectures and mobile computing within the context of business processes and workflow management. The ultimate goal is to incorporate these ideas into IBM's products and prototypes. The project involves IBM groups in Almaden (U.S.A.), Hursley (U.K.), Böblingen (Germany), and Vienna (Austria). In this paper we briefly describe two IBM products, FlowMark, a workflow management system, and MQSeries, a messaging system, as the environments in which we are focusing our research. We also discuss some of our results in the areas of availability, replication, distribution, and advanced transaction models, as well as describe our future research directions.

1 Introduction

During the past few years, there has been a growing interest in workflow applications as a way of supporting complex business processes in modern corporations [Fry94]. The current popularity of business process reengineering has served as an added incentive to further this trend. To address the unique requirements of new applications such as CAD/CAM, CASE, or collaborative editing for which the traditional transaction model is not adequate, and also those related to modeling business processes, i.e., workflows, which involve executing multiple interrelated transactions, potentially on behalf of multiple users, several advanced transaction models have been proposed: Nested Transactions [Mos81], Sagas [GMS87], ConTract model [WR92], Flex Transaction model [ELLR90], and Split-Transactions [PKH88], among others. These models aim at relaxing some of the restrictions imposed by the classical ACID transaction model, thus making it more appropriate for the new applications [Elm92, Moh94]. While much of this work has been done in academic research projects, some industrial groups are also actively working on these topics, with a few of the models having been implemented as prototypes. Unfortunately, currently commercially available products incorporate very few of these results. Even the decade-old nested transaction model is supported only by a handful of products, mostly in a primitive form. As a consequence, the common way of dealing with long running activities is to split them into multiple transactions. This leaves to the application programmer the task of dealing with recovery, transactional dependencies, coordination, and other correctness issues. In contrast to this situation, there are already many workflow products in the market [Fry94] that are being widely used for the same applications targeted by advanced transaction models. Some of the issues we cover in this paper are availability, distributed execution and replication. Most of them are common to any transaction management system, but so far, few of the advanced models have covered these aspects. For this reason, we believe that workflow management is an ideal environment to explore

*This work is partially supported by funds from IBM Hursley (Networking Software Division) and IBM Vienna (Software Solutions Division). Even though we refer to specific IBM products in this paper, no conclusions should be drawn about future IBM product plans based on this paper's contents. The opinions expressed here are our own.

†On sabbatical from the Computer Science Department, University of California at Santa Barbara, CA 93106, USA.

‡Currently visiting IBM Almaden from the Computer Science Department at the University of Massachusetts, Amherst, MA 01003, USA.

advanced transaction management concepts in the context of real user needs [Hsu93]. This work is part of the Exotica project, recently initiated at the IBM Almaden Research Center. The project aims at exploring, and ultimately incorporate the results into IBM's products and prototypes, several research areas from advanced transaction management concepts to client/server architectures and mobile computing within the context of business processes and workflow management. As a first step, we have analyzed workflow systems to identify the requirements and demands of real applications. This has been done in the context of two IBM products, FlowMark, a workflow management system, and MQSeries, a messaging and queuing system. These were obvious choices for us not only because we have access to the code and its developers, but also, particularly in the case of FlowMark, because they are very representative of the state of the art and provide a rich functionality. These products are described in Section 2. Section 3 gives an overview of the research areas covered so far, and Section 4 concludes the paper with a brief discussion of future research directions.

2 FlowMark and MQSeries

In this sections we briefly describe FlowMark, a workflow management system, and MQSeries, a messaging system. FlowMark provides us with a comprehensive framework to reason about workflow and business processes, while MQSeries has been instrumental in developing some of the protocols and designs to increase reliability and availability of the system.

2.1 FlowMark

FlowMark™ , IBM's strategic workflow product, runs across different platforms (AIX, OS/2, Windows). It follows a layered client/server architecture [LA94, LR94] that supports the concepts of *buildtime* and *runtime*, implemented as clients of a FlowMark server. The server, in turn, acts as a client of a centralized database, ObjectStore, where both runtime and buildtime workflow information is stored. At runtime, information flows from the database to the server, from the server to the runtime client, and vice versa. At buildtime, the server remains inactive, with communication between the buildtime component and the database occurring on a client/server basis.

A FlowMark *process* is defined at buildtime and is executed at runtime - in what follows, the word *process* refers to a business process, or workflow, rather than to the conventional operating system process. A process is stored as a schema, copies of which are made to run different instances of the process. There is no dynamic modification of a process' schema, changes to it will not affect instances already running. A process is represented as a weighted, colored, directed acyclic graph (DAG). Nodes in the graph represent *activities* and arcs represent *control* and *data connectors*. Activities can be either simple or subgraph activities. A simple activity has associated with it a program which is invoked when the activity is ready for execution and a qualified user starts it. Subgraph activities correspond to blocks or other processes, and are used to allow nesting and composition in the design of processes. Each activity has a *start condition* and an *exit condition*. The start condition is used to determine when the activity is ready for execution in terms of the flow of control of the overall process. The exit condition determines when an activity has been completed successfully. The order of execution between pairs of activities is specified by means of control connectors. These control connectors evaluate to true or false, and the start condition of an activity is a simple boolean function - AND or OR - of the state of all incoming control connectors for that activity. The true or false value of a connector is determined through its *transition condition*. Control connectors remain unevaluated until the activity at their origin terminates. Once all the incoming control connectors of an activity are evaluated, its start condition is evaluated. Control connectors and the different conditions take care of the flow of control within the process. Data flow occurs through *data connectors* and *input/output data containers*. Each activity has an *input container* and an *output container*. They store the values which are supplied as the input and which are returned as the output, respectively, of the activity's program. The data connectors map values of an output container of one activity to the values of an input container of another activity. There can be data connectors from an activity to itself, which allows the process to iterate over the same activity.

FlowMark supports **forward recovery**, i.e., after a system failure during the execution of a process, FlowMark will resume the execution where it left off. In the case of activities which were in execution at the time

[TM] FlowMark, IBM and OS/2 are trademarks of International Business Machines Corp.

of failure, there are several cases to consider. If the clients executing those activities did not fail, they will keep the information about the execution until the server recovers and then send it the information as usual. If, during the interval while the server recovers, the clients also fail, the information regarding the execution of the activities is lost. FlowMark clients do not use any stable storage to store the results of an activity's execution. In such cases, it is the user's responsibility to resolve the situation by terminating or restarting the activities. Otherwise the server will continue to think that they are in-progress activities.

As a future enhancement, FlowMark needs to support **backward recovery**. This will give the user the ability to request FlowMark to *compensate* (i.e., logically undo) the already-completed activities of an executing process.

2.2 MQSeries and MQI

One way to support asynchronous communication between programs is to use a messaging system. However, to survive failures, such system must provide some form of stable storage where the messages can be kept. This can be done by using recoverable message queues, i.e., persistent queues. This has the advantage that, as messaging is connectionless, it shields the application program from communication failures and recovery. The underlying messaging and queuing infrastructure deals with those problems. Messaging products are referred to as middleware. They interface between applications and communication networks. They shield the complexities of the underlying multivendor, multiprotocol networks, allowing a more user-friendly programming environment in an open, distributed world where interoperability is essential. For this purpose, IBM has defined an application programming interface (API) standard for messaging called Message Queue Interface (MQI) [IBM93]. In addition, IBM has also released a family of products called MQSeries that supports MQI [MD94]. MQSeries products operate on IBM and non-IBM platforms and they support the architected MQI. Communication takes place through named queues that do not require all participating programs to be available, i.e., up and running, simultaneously. Moreover, MQI is not sensitive to network transport protocol differences. It lets the application designer concentrate on the business logic alone rather than having to deal with communications logic also.

An important aspect of this approach is that the interactions between the programs and the message queues occurs in a transactional manner. The two main MQI primitives are *Put*, add a message, and *Get*, retrieve and delete a message. Note that queues can be *local* or *remote*. Put is asynchronous and can be executed in remote and local queues, while get is synchronous and can only be executed over local queues. In the absence of system failures messages obey transaction semantics, i.e., commit and rollback. Get calls can be executed with several options. In case a message is not available, the *wait* option can be used to indicate that the call should wait until a message becomes available. There are several reasons why a message is not available. It may just not be there, it may be there but not committed - a message cannot be retrieved until its corresponding Put operation commits - and it may be held by a Get operation that has not yet committed - a message is not deleted from the queue until the Get operation commits. A Get can also be used to retrieve the next available message in a queue or a message with specific characteristics. Finally, Get can also be used to browse the queue and retrieve a message without deleting it from the queue. When browsing, the *first* and *next* options can be used to look at a sequence of messages - a browse cursor which holds a position in the queue of messages is maintained.

3 Exotica

In this section, we discuss some of the work that we have done in the Exotica project. We have proposed architectures for enhancing FlowMark's availability and scalability characteristics by using MQSeries and multiple ObjectStore servers [AAE+94, AKA+94].

3.1 Message-Based Workflow Management

In designing workflow management systems there is a trend, as we have seen in the case of FlowMark, towards client server architectures in which a dedicated server provides most of the functionality of the system while the computing potential at the clients is barely used. In FlowMark, the execution of complex processes involving several steps across different platforms is controlled from a single ObjectStore server. While there are some aspects of the system that are distributed, an architecture based on a centralized database server is vulnerable to failures of the database server. This approach may not be the most appropriate in certain applications where the

work is divided among many autonomous units and where the transactional volume is too high to be handled in a centralized fashion. In particular, it is not clear how systems of this kind will scale to hundreds of thousands of business processes. Another problem is that a centralized architecture is not only a potential performance bottleneck but also has a single point of failure. If the ObjectStore server were to fail, then the FlowMark components would be unable to make progress once any in-progress activities have finished their execution. This is because the database needs to be accessed to determine what activities are then eligible to execute (i.e., to *navigate* the process graphs). Even activities that are ready to execute cannot be executed until the ObjectStore server recovers. Unfortunately, ObjectStore does not have support for *hot standbys*.

In [AAE+94], we explore the possibility of implementing a fully distributed FlowMark system and discuss the trade-offs of our approach with respect to the current centralized architecture of FlowMark. The key idea of the proposed solution is to eliminate the need for a centralized database by using MQSeries-style persistent messages. The system we propose, *Exotica/FMQM*, FlowMark on Message Queue Manager, is a distributed workflow system in which a set of autonomous nodes cooperate to complete the execution of a process. Each node functions independently of the rest of the system, the only interaction between nodes is through persistent messages informing that the execution of an activity of the process has been completed. With this approach, we avoid the performance bottleneck of having to communicate with a centralized database server during the execution of a process. Moreover, the resulting architecture is more resilient to failures since the crash of a single node does not stop the execution of all active processes. The main contribution of this work is the design of a distributed architecture for workflow management using a persistent message passing system. We have also identified some extensions to the functionality of MQI and MQSeries which would enable the support of FlowMark's rich functionality. For example, in Exotica/FMQM, in order to continue to allow a FlowMark user to log on from different runtime clients, we need the ability to migrate an MQSeries queue from a node to another or to let the user issue Gets involving not only local queues but also remote queues. The latter functionality is also needed to support more than one user being eligible to start the execution of an activity, but still ensuring that only one user is actually allowed to succeed in starting the execution even if multiple of them attempt to start an activity. Currently in FlowMark this kind of synchronization is easily performed because of the use of a centralized ObjectStore database. This is true even if the different users are logged on to different FlowMark servers. The client server support of ObjectStore (e.g., global locking and cache coherency) simplifies the work that needs to be done by the FlowMark servers to accomplish the needed synchronization.

The use of MQSeries for implementing workflow support would mean that some messages may stay in queues for very long times (e.g., weeks) and random retrievals of messages with specific field values would be necessary. These were not the design points for the original MQSeries implementations. So enhancements would be needed to MQSeries if we desire good performance. For example, it should be possible to define indexes on queue contents so that random retrievals do not require sequential scans of queue contents. Another problem with a distributed implementation is that it would be very difficult to support enquiries about the state of a process. Because of the centralized database, today FlowMark supports such functionality very easily.

3.2 Another Approach to Fault-Tolerance and Scalability

In contrast to the more radical approach taken in Exotica/FMQM which involved replacing ObjectStore completely with an MQSeries-based implementation, we also studied the implications of a more evolutionary approach to improving availability and scalability [AKA+94]. The latter approach is to have multiple clusters of FlowMark servers where all the FlowMark servers in a given cluster are all connected to the same ObjectStore server but the different clusters connect to different ObjectStore servers. A FlowMark client logs on to a cluster rather than to an individual FlowMark server. The FlowMark software then assigns the client a particular server to process its requests. The advantage of this is that if the server connected to the client were to fail then the client can be transparently connected to another server. In order to accomplish this, the communication between the FlowMark clients and servers is performed using MQSeries queues rather than using the non-recoverable message support provided by the TelePath component of current FlowMark. To deal with ObjectStore failures, when a client logs on, it is connected automatically to all the clusters in the same fashion described above. By making FlowMark automatically replicate a process definition across the multiple clusters, different instances of the same process can be created in the different clusters. This will let processing load be spread across multiple ObjectStore servers. Under these conditions, if the ObjectStore server in a cluster were to fail, only the process instances running in that cluster will be affected. A client can continue to create new instances of that process

in the remaining clusters and also continue executing activities belonging to instances managed by the surviving clusters.

3.3 Mapping Advanced Transaction Models to FlowMark

In the Exotica project, we have also proposed a way to take user specifications of Sagas or Flexible Transactions and mapping them automatically into FlowMark process schemas [AKA+94]. We believe that this is very useful functionality for workflow modelers using FlowMark. It also shows the modeling power of FlowMark's features. The work involved in performing the mapping is identifying the appropriate modeling constructs (e.g., simple activities or a block of simple activities) and appropriate checking of conditions (e.g., exit, entry and transition conditions) for each of the advanced transaction models. Also, output containers need to be used to keep track of history of execution of activities so that when there is a need to completely or partially abort the computation, the values in the containers can be used to determine what activities actually executed and hence need to be compensated. The advanced transaction models that we can handle in this fashion are ones which do not involve making changes to resource managers accessed by the FlowMark activities' programs. For example, nested transactions and split transactions cannot be handled with this approach since those require changes to the concurrency control and recovery mechanisms. FlowMark is neither aware of nor has any control over the concurrency control and recovery features of the resource managers accessed by the activities.

3.4 Mobile Computing and Disconnected Operation

Another area which we are exploring is the support for mobile computing. The problem today is that all the information relating to an executing process is kept in the ObjectStore database and is accessible only to the servers. The logic for navigating the process graph is present only in the FlowMark servers. As a result, once a client finishes an activity it has to talk to the server before it can make further progress (e.g., execute the follow-on activity to the one that was just completed). Also, once an activity's execution is begun, the client should not be powered off before it finishes executing the activity and communicating the results to the server. Otherwise, the problems that we discussed before will crop up. We need to change some of these features of FlowMark to allow mobile clients to check out a subgraph of activities for disconnected execution and to make them record in stable storage the results of executing activities.

4 Future Work

Since FlowMark is unaware of what happens when an activity executes is that the commit of that activity must be an independent action that precedes the recording of the completion of that activity in the ObjectStore database. Hence, a failure could occur in such a way that the activity commits but the recording of that activity's completion does not occur in the database because the client executing the activity fails. We call this the *gap problem*. The gap problem is not unique to FlowMark. Problems like this exist with most of the proposals in the research literature. As was pointed out in [MD94], the problems are severe when pre-existing applications have to be accommodated. Some of these applications may even execute more than one ACID transaction in a single invocation of an application.

As we mentioned before, the FlowMark client has *amnesia* when a failure occurs. In the above scenario, when the client recovers, human intervention would be needed to resolve the situation where the server continues to think that the activity is still in execution. There may not be a simple way for even the human being to determine whether or not the activity's transaction had committed before the failure. Even if the human being were to determine that in fact the transaction had committed, he may not be able to find out the values for the variables in the output container. In order to make it possible to do the latter, the application probably would have to be modified to make it insert into a database the values intended for the output container. This insert would have to be committed atomically along with any business-specific recoverable update operations that the application performs.

It is not easy to take care of the above problems automatically. We will be exploring some alternatives to provide better support in this area. The primary problem is that ObjectStore does not support the prepare interface (e.g., X/Open XA protocols). Secondly, even if such an interface were to exist, if the activity's application were to be a pre-existing one, then changes would need to be made in it to make it participate in a distributed

transaction involving its resource managers and FlowMark. This assumes that the resource managers updated by the application are capable of handling two-phase commit with an external coordinator. An alternative is to make use of MQSeries and not require ObjectStore to support the prepare interface. This means that MQSeries has to support the prepare interface. This may also involve changes to the application to make it interact with MQSeries or at least avoid issuing the commit call before returning to the FlowMark client code. If the commit call is not explicitly issued by the application, as it is typically the case in CICS applications, then the FlowMark client code could place the values of the output container in a message, insert the message into a queue and then commit the whole transaction.

An even more sophisticated degree of availability support that we wish to explore is being able to let the state of a process tracked in an ObjectStore server be replicated across one or more other ObjectStore servers. ObjectStore itself does not provide a *hot standby* support. Hence, we need to perform the replication at a higher level. This could be accomplished by making the FlowMark server perform state changes for a process in an ObjectStore database, the primary database, and communicate those updates to other FlowMark servers, the backup servers. The backup servers can then make the corresponding updates in their ObjectStore databases, the backup databases). Another solution is to let the primary FlowMark server itself perform the updates directly in the backup ObjectStore databases. The communication between FlowMark servers can be performed using MQSeries to make it reliable and asynchronous. One problem here is that ObjectStore does not have a (two-phase commit) *prepare* interface and hence the changes to the ObjectStore database and MQSeries queues cannot be guaranteed to be performed atomically. Since we are dealing with an object-oriented DBMS in which updates to persistent data are performed by making ordinary C++ operations, exactly how to communicate the updates is not clear. Having a high-level language like SQL for updates would have made the job somewhat easier. One possibility is to send to the backup node what is contained in the TelePath messages sent by FlowMark clients to FlowMark servers on the termination of an activity or when a user tries to execute a ready activity. Approaches like the one in [MTO93] in which log records are used to maintain a backup copy of a database are probably not easy to follow here since they would require changes to the internals of ObjectStore. Also, the logging done in ObjectStore is very physical in nature (i.e., there is no operation logging). So it is next to impossible to decipher which objects actually got modified by examining a log record. This capability is important if we want to perform replication only in a few cases due to the cost involved.

Recently, a proposal, called Spheres of Joint Compensation (SJCs), for providing backward recovery in FlowMark has been made [Ley95]. We are studying the implications of this proposal on the externals and internals of FlowMark. An SJC can be thought of as a business transaction (e.g., like a saga). It consists of a collection of FlowMark activities (simple and/or non-simple). The activities need not even be contiguous in the process graph. The proposal permits compensating activities to be defined with respect to individual activities as well as for collections of activities. An SJC can be aborted by compensating its activities individually or by invoking the compensation activity for the sphere as a whole. There are many other options associated with a sphere, e.g., restarting a sphere's execution and so on. We are studying the semantics of this powerful concept and its applicability to different application scenarios.

Acknowledgements We would like to thank the following current or former IBM employees in Almaden, Böblingen, Hursley, and Vienna for valuable discussions on advanced transaction models and workflow in general and on FlowMark and MQSeries in particular: Wolfgang Altenhuber, Dick Dievendorff, Mike Halperin, Robert Junghuber, Frank Leymann, Peter Lupton, Susan Malaika, and Walter Taus.

References

[AAE+94] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör, and M. Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. Research Report RJ 9912, IBM Almaden Research Center, November 1994.

[AKA+94] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Failure Handling in Large Scale Workflow Management Systems. Research Report RJ 9913, IBM Almaden Research Center, November 1994.

[ELLR90] A.K. Elmagarmid, Y. Leu, W. Litwin, and M.E. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. In *Proc. of the 16th VLDB Conference*, August 1990.

[Elm92] A.K. Elmagarmid, editor. *Transaction Models for Advanced Database Applications.* Morgan-Kaufmann, 1992.

[Fry94] C. Frye. Move to Workflow Provokes Business Process Scrutiny. *Software Magazine*, pages 77–89, April 1994.

[GMS87] H. García-Molina and K. Salem. Sagas. In *Proc. 1987 SIGMOD International Conference on Management of Data*, pages 249–259, May 1987.

[Hsu93] M. Hsu. Special Issue on Workflow and Extended Transaction Systems. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 16(2), June 1993.

[IBM93] IBM. *Message Queue Interface: Technical Reference.* IBM, April 1993. Document No. SC33-0850-01.

[LA94] F. Leymann and W. Altenhuber. Managing Business Processes as an Information Resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[Ley95] Frank Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *GI-Fachtagung Datenbanken in Büro Technik und Wissenschaft - BTW'95*, Dresden, Germany, March 1995. Springer Verlag.

[LR94] F. Leymann and D. Roller. Business Processes Management with FlowMark. In *Proc. 39th IEEE Computer Society Int'l Conference (CompCon), Digest of Papers*, pages 230–233, San Francisco, California, February 28 – March 4 1994. IEEE.

[MD94] C. Mohan and R. Dievendorff. Recent Work on Distributed Commit Protocols, and Recoverable Messaging and Queuing. *Bulletin of the Technical Committee on Data Engineering*, 17(1):22–28, March 1994. IEEE Computer Society.

[Moh94] C. Mohan. Advanced Transaction Models - Survey and Critique, 1994. Tutorial presented at ACM SIGMOD International Conference on Management of Data.

[Mos81] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, M.I.T. Laboratory for Computer Science, Cambridge, Massachusetts, MIT Press, 1981.

[MTO93] C. Mohan, Kent Treiber, and Ron Obermarck. Algorithms for the management of remote backup data bases for disaster recovery. In *Proceedings 9th International Conference on Data Engineering*, Vienna, Austria, April 1993.

[PKH88] Calton Pu, Gail E. Kaiser, and Norman Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings 14th Conference on Very Large Data Bases (VLDB)*, pages 26–37, Los Angeles, California, 1988.

[WR92] H. Waechter and A. Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–263. Morgan Kaufmann Publishers, San Mateo, 1992.