Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management

G. Alonso, C. Mohan, R. Günthör
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120, USA.
E-mail: {gustavoa,rgunther,mohan}@almaden.ibm.com
D. Agrawal, A. El Abbadi
Computer Science Department, UC Santa Barbara,
Santa Barbara, CA 93106, USA.
E-mail: {agrawal,amr}@cs.ucsb.edu
M. Kamath
Computer Science Department, UM at Amherst,
Amherst, MA 01003, USA.
E-mail: kamath@cs.umass.edu

Abstract

In the past few years there has been an increasing interest in workflow applications as a way of supporting complex business processes in modern corporations. Given the nature of the environment and the technology involved, workflow applications are inherently distributed and pose many interesting challenges to the system designer. In most cases, a client/server architecture is used in which knowledge about the processes being executed is centralized in one node to facilitate monitoring, auditing, and to simplify synchronization. In this paper, we explore a novel distributed architecture, Exotica/FMQM, for workflow systems in which the need for such a centralized database is eliminated. Instead, we use persistent messages as the means to store the information relevant to the execution of a business process. Our approach is to completely distribute the execution of a process so individual nodes are independent. The advantages of this approach are increased resilience to failures and greater scalability and flexibility of the system configuration.

To appear in **Proc. IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations**, Trondheim, August 1995.

Keywords

Workflow Management Systems, Distributed Systems, reliability, scalability.

1 INTRODUCTION

Business processes within large organizations often involve a large number of resources, people and tools distributed over a wide geographic area (in what follows, the word *process* refers to a business process, or workflow, rather than to the conventional operating system process). *Workflow management systems* are used to automate the coordination of all these diverse elements thereby increasing the efficiency of the corporation and its ability to monitor its activities. In designing such systems there is a trend 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. There are a number of reasons for this choice: lightweight clients, centralize monitoring and auditing, simpler synchronization mechanisms, and overall design simplicity. However, an architecture based on a centralized server is vulnerable to server failures and offers limited scalability due to the potential performance bottleneck caused by the centralized server. Hence, client/server architectures may not be the most appropriate approach in certain applications. In particular, it is not clear how this kind of systems will scale to hundreds of thousands of business processes or will be able to provide the high availability required in mission critical applications. In this paper, we explore a fully distributed workflow system and discuss the trade-offs of our approach with respect to a centralized architecture. The key idea of our approach is to eliminate the need for a centralized database by using persistent messages. We use the *workflow reference model* proposed by the Workflow Management Coalition [Hollinsworth, 1994] to guarantee the generality of the results. The Workflow Management Coalition is formed by a group of vendors setting standards for the interoperability of their systems. Our immediate research, however, has been centered around FlowMark [IBM, 1995d], [IBM, 1995a], [IBM, 1995b], [IBM, 1995c], [Leymann and Altenhuber, 1994], [Leymann and Roller, 1994], IBM's workflow product, and MQI [IBM, 1993], [Mohan and Dievendorff, 1994], an IBM defined API for persistent messaging, along with *MQSeries*, a family of products that supports MQI. This has allowed us to check our assumptions and results against a real system and real customer requirements. Both FlowMark and MQI follow closely standards in their respective areas, making the issues discussed in this paper pertinent to workflow management systems in general.

The system we have designed, *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 a step of the process has been completed, thus avoiding the performance bottleneck of having to communicate with the 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. Our main contributions are to present a distributed architecture for workflow management using a persistent message passing system and provide an in-depth analysis of the impact a fully distributed architecture has on the workflow model.

*IBM Almaden Research Center's Exotica project aims at incorporating advanced transaction management capabilities in IBM's products and prototypes. Exotica also involves IBM groups in Hursley (U.K.), Böblingen (Germany), and Vienna (Austria).

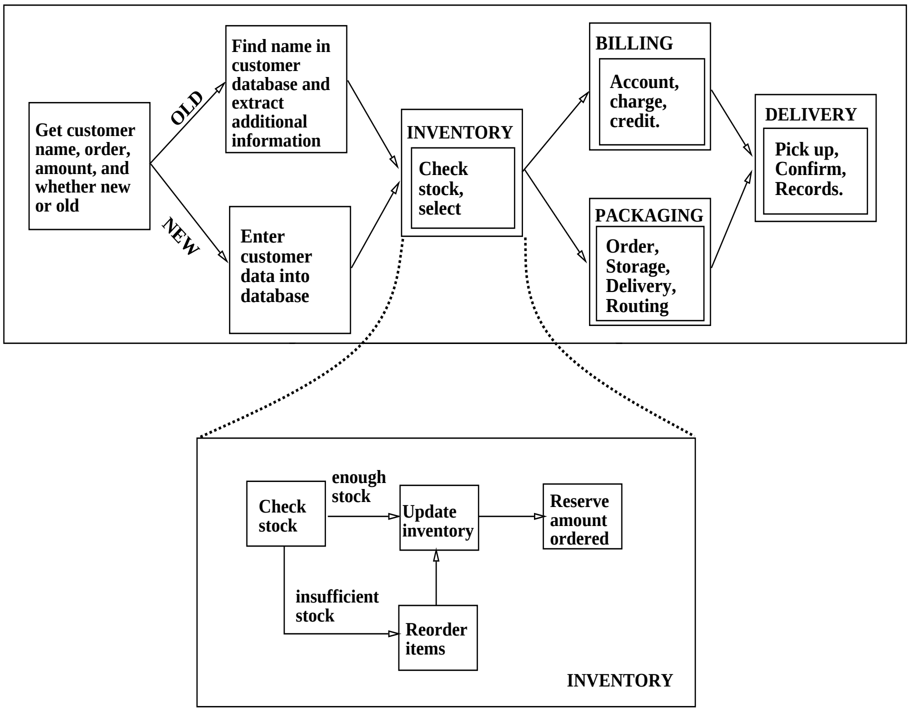

Get customer name, order, amount, and whether new or old
OLD
NEW
Find name in customer database and extract additional information
Enter customer data into database
INVENTORY
Check stock, select
BILLING
Account, charge, credit.
PACKAGING
Order, Storage, Delivery, Routing
DELIVERY
Pick up, Confirm, Records.
Check stock
enough stock
Update inventory
Reserve amount ordered
insufficient stock
Reorder items
INVENTORY

Figure 1 A business process example.

The paper is organized as follows: in the next section, we describe the model proposed by the Workflow Management Coalition and some of the particular details of FlowMark, as well as a generic persistent message mechanism. In Section 3, we present a novel architecture for workflow systems based on persistent message passing. Section 4 discusses practical aspects of the implementation and the impact of distribution on the overall model. In Section 5 we briefly discuss related work. Section 6 concludes the paper.

2 BACKGROUND: WORKFLOW AND PERSISTENT MESSAGING

To establish a common notation, in this section we will present the reference model of the Workflow Management Coalition as well as a generic persistent message mechanism. To further focus the discussion and cover some implementation details, FlowMark is also briefly described.

2.1 Workflow Systems

The work taking place in large organizations is often referred to as *business processes*. A business process is "a procedure where documents, information or tasks are passed between participants according to defined sets of rules to achieve, or contribute to, an overall business goal" [Hollinsworth, 1994]. Figure 1 shows an example of a nested business process [García-Molina et al., 1990]. In the context of information systems, these business processes need a suitable representation. This representation is called a *workflow*. Hence, a *workflow management system*, WFMS, is "a system that completely defines, manages

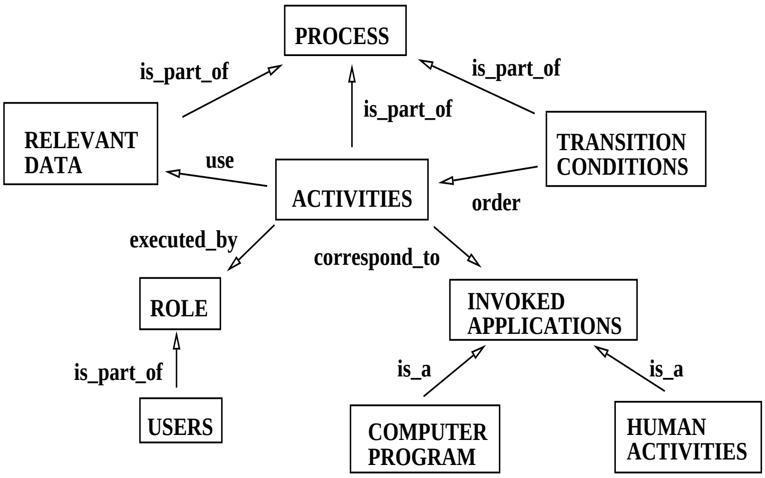


Figure 2 Elements of the Workflow Model.

and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic" [Hollinsworth, 1994].

In more concrete terms, the task of the WFMS is to schedule the execution of activities. In scheduling these activities, the WFMS determines what to execute next, locates the tools associated with each activity, transfers information from activity to activity, assigns activities to users, checks the timeliness of the execution, monitors the overall progress and determines when the process has been completed successfully.

2.2 Workflow Model

To perform its task, the WFMS needs to be supplied with a description of the business process. This is a workflow model. The Workflow Management Coalition has proposed the following meta-model [Hollinsworth, 1994]. A business process is represented as a *process* with a name, version number, start and termination conditions and additional data for security, audit and control. A process consists of *activities* and *relevant data*. Each step within a process is an *activity*, which has a name, a type, pre- and post-conditions and scheduling constraints. It also has a *role* and one *invoked application* associated with it. The *role* determines who will execute the activity. The *invoked application* is the tool to be used in the execution, which may range from computer programs to human activities with no specific tool, e.g., a meeting. Furthermore, activities use *relevant data* defined as the data passed to and produced by the activities. Relevant data includes name and type, as well as the path specifying the activities that have access to it. The flow of control within a process - what to execute next - is determined by *transition conditions*, usually boolean expressions based on relevant data. The users of the system are represented in terms of *roles*. Invoked applications have a name, type, execution parameters and, in the case of programs, a location or access path. All these elements and their relationships are shown in Figure 2, which is our own interpretation of the reference model [Hollinsworth, 1994].

2.3 Navigation and Execution in FlowMark

The reference model does not provide implementation details. To discuss how a process is actually executed we need to resort to an existing system. We use FlowMark since its model closely follows the reference model of the Workflow Management Coalition. In FlowMark, omitting users and invoked applications, an activity has the following concepts associated with it:

- **Flow of Control**: specified by *control connectors* between activities, is the order in which these activities have to be executed. This corresponds to the *transition conditions* of the reference model.
- **Input Container**: a sequence of typed variables and structures which are used as input to the invoked application.
- **Output Container**: a sequence of typed variables and structures in which the output of the invoked application is stored.
- **Flow of Data**: specified through *data connectors* between activities, is a series of mappings between output data containers and input data containers to allow activities exchange information. The data containers and the flow of data form the *relevant data* mentioned the reference model.
- **Start Condition**, that must be met before the activity is actually scheduled. If the condition can never be met, then the activity is marked as having actually terminated. The start condition is a boolean expression involving the states of the incoming control connectors. It corresponds to the pre-condition of the *activity* in the reference model.
- **Exit Condition**, that must be met before an activity is considered to have completed. If the exit condition is not met, then the activity will be rescheduled for execution. It corresponds to the post-condition of the *activity* in the reference model.

Navigation in FlowMark takes place through the data and control connectors. Control connectors determine the flow of control within the process. A control connector between two activities, t_1 and t_2, implies that t_2 can only start execution after t_1 has terminated. Thus, when t_1 terminates, t_2 must be notified of the fact - note that the user written code for t_2 does not get executed at this point, the notification is internal to FlowMark. On the other hand, if t_1 is never going to be executed, t_2 must also be notified of that fact. The notification will be done by t_1 upon termination or when it is marked as finished because it will not be executed. The process by which activities that will never execute are marked off is known as *dead path elimination* [IBM, 1995a], [IBM, 1995b], [Walk, 1993].

Each control connector has a state. Before the activity from where it originates finishes, the control connector is *unevaluated*. Once the activity terminates, the control connector is evaluated and its state is set to TRUE or FALSE. The evaluation involves checking whether the *transition condition* for the connector is true or false. Such transition conditions are boolean expressions involving data returned by activities in the output containers.

Similar to the flow of control, FlowMark lets the user define the flow of data in a given process. The flow of data is expressed via data connectors. Each data connector is a mapping between the output container of an activity and the input container of another. To avoid race conditions, the data flow must follow the ordering imposed by the control flow, except for the fact that an activity can have a data connector between

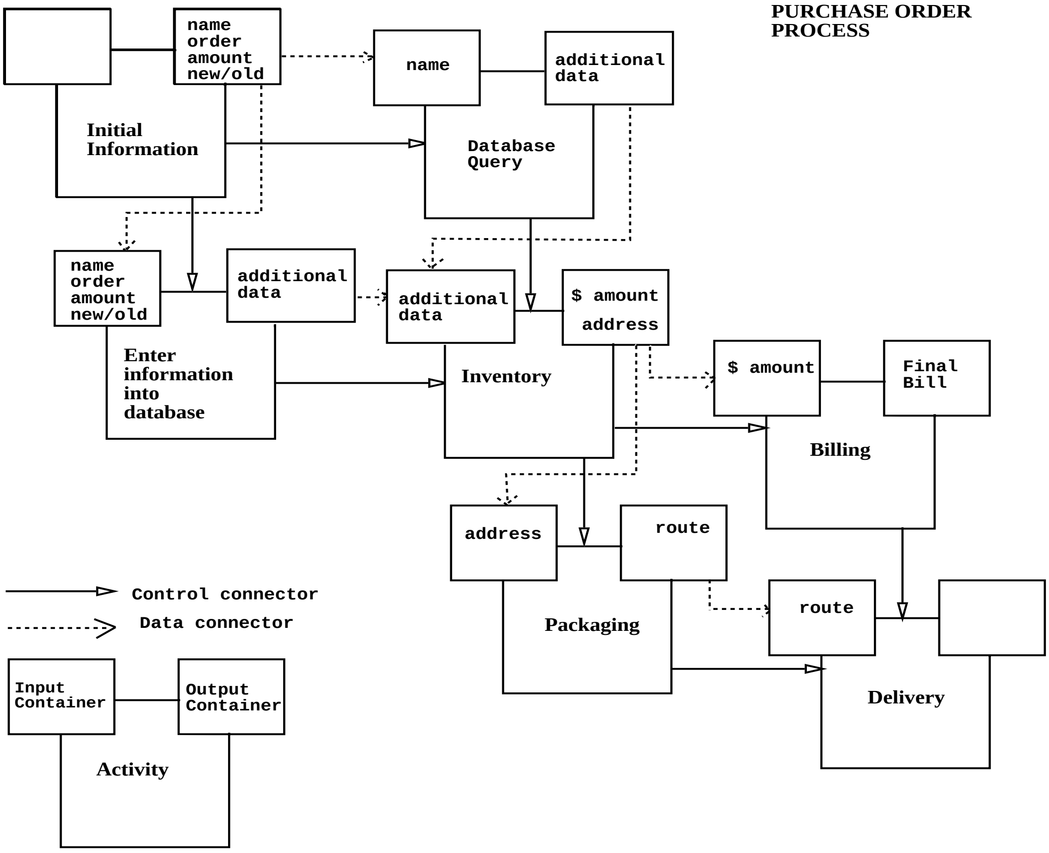


Figure 3 The example of Figure 1 formalized using FlowMark concepts.

its output container and its input container to provide feedback information in loops. Figure 3 illustrates the purchasing example discussed earlier in the context of FlowMark and illustrates the usage of the FlowMark entities discussed above. Finally, processes are independent entities, i.e., there are no inter-process dependencies.

2.4 The Architecture of FlowMark

FlowMark runs across different platforms (AIX, OS/2, Windows) and supports distribution of most of its components. However, in its current version, persistent data resides in a single database server, ObjectStore. This has facilitated the design of the overall system, but introduces a single point of failure in the architecture. As far as we know, most existing systems are also based on a centralized database and, hence, suffer from the same problem. Besides the database server, FlowMark is organized into three other components: *FlowMark Server*, *Runtime Client* and *Buildtime Client*. These correspond to the three functionalities mentioned in the reference model [Hollinsworth, 1994]: Buildtime Client for build-time functions, Runtime Client for run-time interactions, and FlowMark Server for run-time control. A FlowMark system can have several of these components executing at the same time. A FlowMark server is connected as a client to a database. FlowMark servers can reside in remote hosts other than the one where the database is located. Communication between the server and the clients takes place through TCP/IP, NetBIOS or APPC, depending on the platform.

Navigation, as explained above, takes place in the FlowMark server. The creation and definition of processes is done using the GUI interface provided by the buildtime clients. Finally, users and programs interact with the server through the runtime clients. There can be several clients, both buildtime and runtime, attached to a server.

2.5 Persistent Messages

In the past few years, there has been a growing interest in message passing/queuing [Mohan and Dievendorff, 1994]. In general, many applications require a communication mechanism that hides the complexities of the underlying system and allows communications across heterogeneous platforms and independently of the protocol used. These services are, preferably, connectionless and accessible through API calls. The most common mechanism is to provide a local queue where applications can place and retrieve messages. Once left in the local queue, the communication system will take care of delivering it to the appropriate queue in a remote machine. This has the advantage of being protocol independent since the application just uses queues. Moreover, these queues can be made persistent so messages survive crashes, making asynchronous communication possible between applications that run at different points in time. There are even some standarization efforts underway lead by the *Message Oriented Middleware Association*, MOMA. Numerous products exist: DEC's *MessageQ*, Transarc's *Recoverable Queuing Services*, Novell's *Tuxedo/Q*, and IBM's MQSeries, to name a few. The type of features provided by these products is very suitable for connecting highly distributed applications, therefore we will rely on the features they provide.

We base our design on a generic queue system with the following characteristics. Nodes will communicate via recoverable queues, i.e., once a message is placed on a queue and the corresponding transaction is committed it becomes persistent until it is retrieved. Each queue is associated with one node in the system. Messages are added and retrieved from a queue using *PUT* and *GET* calls, which can be issued both on local and remote queues. GET calls can access any message in a queue, not necessarily only the first message in the queue. Finally, it is possible to browse through a queue without actually retrieving any messages. A key aspect of the messaging system is its ability to support transactional semantics for both PUT and GET calls. We will use this property to ensure that messages do not get lost when failures or crashes occur.

3 THE DISTRIBUTED ARCHITECTURE OF EXOTICA/FMQM

In this section, we describe a distributed architecture for a workflow management system with high reliability and scalability. We refer to this system as *Exotica/FMQM* - FlowMark on Message Queue Manager.

3.1 Motivation

As we have already pointed out, the centralized database needs to be replaced in the new architecture, both to eliminate it as a single point of failure and a bottleneck. In a workflow system, the execution of a business process is analogous to a distributed computation. That is, each node executes its part of the job and synchronizes with other nodes by communicating over a network. However, in our case we cannot afford node or communication failures affecting the work in the entire system. One more restriction is that the nodes, the runtime clients, must be kept simple since they probably will run in low-end machines like PCs or laptops, which rules out the possibility of providing each node with full database functionality. To overcome these limitations, we propose to use a persistent message mechanism and a minimal amount of persistent storage at each node. The communication layer is used as persistent storage to maintain workflow related information. The storage at each node is used to recover from failures. Moreover, we have tried to minimize the need for storage outside of message queues at each node so it can be easily implemented in files.

With this approach, a process is executed in a truly distributed fashion, with each node holding only the information it needs to perform its part of the workflow. Failures can only lead to blocking behavior on the activities that are being executed at the failed node but single failures do not prevent other nodes from working on different parts of the same processes or in completely different processes.

For notational purposes, a process refers to the representation of a business process. There can be many instances of the same process running concurrently, these are called process instances. Note that some of the information stored at each node will relate to the process - static information - while other parts will relate to process instances - runtime information.

3.2 Process Definition and Binding

According to the reference model, all the procedures related to defining a process are performed in the *Buildtime Client*, an independent module that provides a graphical user interface to perform all these tasks. The equivalent to the Buildtime Client in Exotica/FMQM is any node that has the appropriate user interface and can check the process definition for inconsistencies and errors. We will refer to this node as the *process definition node*. The nodes where the process instances are executed are the *runtime nodes*. For simplicity, we will refer to them simply as nodes.

Once a process has been defined, this definition is compiled to determine the information relevant to each node, binding activities to the nodes where they will be performed. At each node, there is a *node manager* in charge of communicating with the process definition node. The node manager maintains a *process table* on a per process basis, which contains the static information related to the execution of instances of each process. During the execution of process instances, communication between nodes takes place through queues. At each node there will be a queue for messages related to all the instances of the same process. Queues are named with the process id and the node id. Hence, each process table corresponds to a queue, and both correspond to a single process. Each node is interested in the part of the process that it will have to execute. Therefore, the process definition node sends to each involved node's manager only the information pertaining to the activities that will execute at that node. The node manager, upon receiving the information from the process definition node, creates the process table.

Once a node manager has all the static information stored in the process table, it starts a thread that will be in charge of coordinating the execution of instances as they arrive at the node. We will refer to this as the *process thread.* The first task of the process thread is to create the queue where the messages from other nodes will be received. A process thread examines the queue for incoming messages and decides when an activity is ready for execution. When this happens, an *activity thread* is started to manage the execution of the activity.

The process table is a list of what to do to execute the activities that correspond to the node. Hence, it is organized according to these activities. Without getting into details of how its implementation could be optimized, the process table has an entry for each activity in the following form:

- Process identifier.
- Activity identifier.
- List of incoming control connectors, plus the corresponding starting condition.
- List of outgoing control connectors, plus the associated queue where the result of the evaluation of the connector is reported. For instance: $t_4, Q_{D,0}, CONDITION$ represents activity t_4, of process 0, located in node D, and accessible through a queue named $Q_{D,0}$. $CONDITION$ is a transition condition, i.e., a boolean expression used to decide whether to follow or not the path marked by the connector.
- Exit condition, which is a boolean function of the values found among the output data.
- Incoming data connector, plus the filtering information to select the relevant data. For instance, $t_1, FILTER$. $FILTER$ specifies which is the relevant data out of all the data received. Since many activities may read data from the same output data container, each one of them needs a mapping to extract only the relevant data. The filter specifies this mapping.
- Outgoing data connector, which specifies the receiving activity and the queue where this has to be reported. For instance, $t_4, Q_{D,0}$.
- Input data template, where the type of the input data is specified.
- Output data template, where the type of the output data is specified.

To summarize, what we have described so far is the per-process data structures, queue, threads and algorithms at each node. The sequence of events is as follows: a user first creates a process. The process is compiled in the process definition node. After compilation, the process is divided in several parts and each part is distributed to an appropriate node. The division of the process into parts will be based on the users associated with the different nodes and the roles associated with the different activities in the process. Upon receiving its part of the process, a node creates a process table to store this information and starts a process thread to handle the execution of instances of such process. Finally, the process thread creates a queue for communicating with other nodes all information relevant to instances of the process. Figure 4 shows all these components and their interactions. Specific examples of process tables are shown in Figure 5.

3.3 Process execution

During execution, the information regarding the process instances will be stored in an *instance table* handled by the activity threads. These threads are responsible for the execution of individual activities within an instance. All PUT and GET calls are executed within transactions, i.e., their effects are not permanent until the transaction commits.

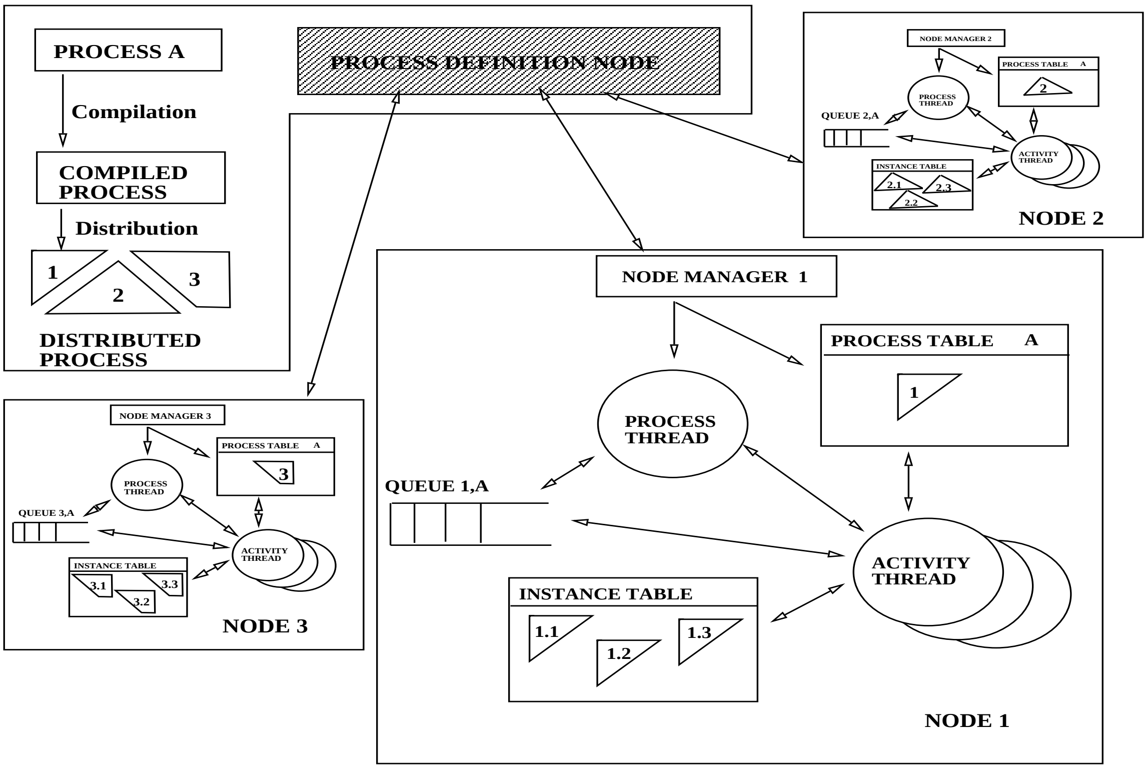


Figure 4 Distribution of a process among several nodes with three active instances.

When a process instance is started, for all starting activities of that instance - those without predecessors, i.e., without incoming control connectors - messages are sent to the nodes where the activities are located. This is done by PUTting the messages in the appropriate queues. In a node receiving that message, the sequence of activities that take place is as follows. The process thread scans the queue periodically, looking for new messages. If there is a triggering facility in the queue system it can be used to receive messages more efficiently. When it finds a new message, it browses through it - without retrieving it from the queue, this is important in case of failures - and spawns an activity thread. The activity thread first checks the process table to find the data pertaining to that activity. With this data it creates an entry for that activity in the instance table where it will record the progress of the activity. It then checks the start condition of the activity. If it cannot be evaluated, because not all incoming control connectors have been evaluated, it makes a note in the instance table about what control connector has been evaluated - the one to which the message received corresponds - and goes to sleep until the process thread awakens it again with new messages.

If the start condition is false, then the activity is considered terminated. Dead-path elimination messages are sent along the outgoing control connectors - PUT calls to the corresponding queues - the entry is removed from the instance table and the messages corresponding to this activity are retrieved from the process queue - with a GET call. All the operations performed upon termination of the activity are one atomic action. This can be accomplished by using the transactional properties of the GET and PUT calls.

If the start condition evaluates to true, then the corresponding application is invoked, passing to it the data in the input template. When the application terminates, it returns an output data template which is stored in the instance table. Then the activity thread evaluates the exit condition of the activity. If it is false, the application is invoked again. If it is true, the activity is considered terminated. Upon successful termination, the outgoing control connectors' CONDITIONs are evaluated and messages are sent to the appropriate nodes with PUT calls to their queues. The output data template is sent to the nodes specified in the process table as outgoing data connectors. Then the entry is removed from the instance table and all messages corresponding to the activity are retrieved from the process queue. Note that each message corresponds to one activity, once the activity successfully terminates there is no need to keep the messages. These operations are performed as an atomic transaction as explained above.

Note that some of the messages received by a node may correspond to data templates sent by other nodes. In this case, the activity thread uses the FILTER specified in the process table for that data connector to extract the relevant information. This information is then stored in the instance table, in the input data template entry for that activity.

The instance table contains runtime information concerning the evolution of activities as different instances are executed. It has the following structure:

- Process and instance identifiers.
- Activity identifier.
- Input data, the actual values for the input data. This will be gradually filled as the data becomes available - as messages containing the data arrive.
- Output data, the actual values of the output data. This is what will be sent to all activities specified in the outgoing data connectors.
- Status of the activity: *inactive*, *in execution*, *executed*, *terminated*.
- Start condition, to store the partial results necessary to evaluate the start condition. As messages are received, the different parts of the boolean expression are evaluated.

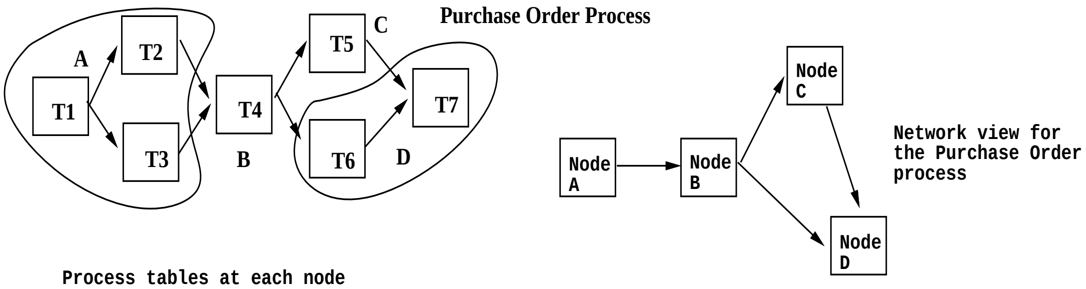
The instance table is stored in memory, there is no need to keep it in stable storage. The process table, however, needs to be in stable storage to allow proper recovery after failures. We will elaborate more on these points in the discussion.

3.4 Example

To better illustrate the above concepts, the example described in the initial sections can be adapted to the new notation. Assume that there are four nodes, A, B, C, and D, involved in the processing of a purchase order. The process table for this process at each node is shown in Figure 5. For simplicity, we have omitted in this figure the CONDITION of the control connectors, the exit conditions of the activities, and the FILTER of the incoming data connectors.

Note that in the process table, the information is expressed in terms of actual queues. The notation used is as follows. $T2Q_{A,0}$ - outgoing control connector of node A - means that there is a control connector between T_1 and T_2 for process 0, i.e., node A upon completing the execution of activity T_1, must notify the node who "owns" activity T_2. This can be found in the queue name, $Q_{A,0}$, which specifies that such node is A, reachable through queue $Q_{A,0}$. This queue name will be the one used in PUT calls.

As an example of what happens during runtime, assume activity T_4 is executing at node B. When this activity terminates, let's assume it is successful, its activity thread has to



Process tables at each node

Node A
Process Table

Proc.	Act.	In.CC	Out.CC	Exit	In.DC	Out.DC	I/Data	O/Data
P0	T1	_	T2QA0 T3QA0	_	_	_	_	name order amount new/old
P0	T2	T1QA0	T4QB0	_	T1QA0	T4QB0	name	_
P0	T3	T1QA0	T4QB0	_	T1QA0	T4QB0	name order amount n/old	addit. info.

Node B
Process Table

Proc.	Act.	In.CC	Out.CC	Exit	In.DC	Out.DC	I/Data	O/Data
P0	T4	T2QA0 T3QA0	T5QC0 T6QD0	_	T2QA0 T3QA0	T5QC0 T6QD0	add. info.	$amount address

Node C
Process Table

Proc.	Act.	In.CC	Out.CC	Exit	In.DC	Out.DC	I/Data	O/Data
P0	T5	T4QB0	T7QD0	_	T4QB0	T7QD0	$amount	Final bill

Node D
Process Table

Proc.	Act.	In.CC	Out.CC	Exit	In.DC	Out.DC	I/Data	O/Data
P0	T6	T4QB0	T7QD0	_	T4QB0	T7QD0	address	route
P0	T7	T5QC0 T6QD0	_	_	T6QD0	_	route	

Figure 5 The example of Figure 1 distributed among four nodes and their corresponding process tables.

send the output data template along the outgoing data connectors, and notify about the termination along the outgoing control connectors. All this is done with PUT calls. For the control connectors, it will look in the *Out.CC* column of its process table for activity T_4. There it finds the name of the next activities, T_5 and T_6, and the queues to which it has to send the messages, $Q_{C,0}$ and $Q_{D,0}$. Hence, it will perform one PUT call to each of these queues. For the data connectors, the mechanism is similar. In the column *Out.DC* of the T_4 entry in the process table of node B, there are two activities indicated, T_5 and T_6, and their queues, $Q_{C,0}$ and $Q_{D,0}$. The output data template is sent to those nodes by PUTting it in their queues.

4 DISCUSSION

In distributing the execution of a process completely, we had to make certain assumptions that are not necessary in a client/server architecture. For instance, with a centralized server each process has an owner, who can start process instances, abort their execution, and is notified of their termination. When the process is distributed among many nodes, the notion of a process owner is much less obvious. For simplicity, we assume that a process instance starts by executing an activity that triggers the execution of all starting activities of the process instance. This is similar to the notion of "process owner", in the sense that the user assigned to the starting activity is the one who can effectively start an instance of the process. Similarly, a process instance is deemed terminated when all of its activities have terminated or have been marked as such by dead path elimination. In a client/server architecture all this information is centralized in the database, and hence checking for the process instance termination event is simple. In a distributed system, where the status of each activity is stored in a different node, process termination could raise some complicated issues similar to those of snapshots and termination detection [Chandy and Lamport, 1985], [Misra, 1983]. To avoid such complexity, we assume that all processes have a *terminal* activity such that when this activity terminates, or it is marked as terminated, the process has also terminated. This terminal activity is unique within a process and follows all other activities in that process. For notification purposes, this activity will have to contain information about the user that needs to be notified of the termination of the process instance. Similar problems arise for auditing and monitoring the system.

Another interesting aspect of the distributed architecture is the management of worklists. Worklists are lists of workitems belonging to one user. In a centralized system, this is very easy to maintain since users only need to logon to the central server to retrieve their worklist. Once retrieved, the server can update it by sending the updates to the runtime client where the user is currently logged on. Moreover, an activity may appear on several worklists simultaneously, but only one user will be allowed to execute it. The synchronization problem of ensuring that only one user actually executes the activity is solved by having the server select the user who contacts it first. These two features are complex to implement in a fully distributed environment since activities are associated with nodes. The use of remote GET helps to allow an activity to appear in several worklists. Nodes will be notified that messages are in a queue, rather than getting the message themselves. The first node to GET the message from that queue will be the one to which the activity is assigned. Other nodes trying to GET the message will find that it has already been retrieved from the queue and therefore discard the activity.

To achieve resilience to failures we use transactional PUT and GET calls, plus the ability to browse a queue without retrieving messages. When a message is received regarding an activity, it is not retrieved from the queue until the activity has terminated. In other words, messages are kept until they are not needed anymore, and they are removed in an atomic operation. In this way, in case of failures which result in the loss of the instance table, a recovering node will restart the process thread which will find the messages still intact in the queue and, based on them, spawn again the appropriate activity threads which will reconstruct the instance table. This is the reason why instance tables do not need to be stored in stable storage and why process tables should be kept in stable storage. For the sake of completeness, it must be mentioned that an audit log is necessary to keep track of which activities have completed execution and to store messages, in case the user decides to restart a finished activity. This audit log can be another queue, checkpointed every so often.

Persistent messages may be too expensive in same cases. A system designer should consider to use both *persistent* and *non-persistent* messages. Persistent messages survive crash failures whereas non-persistent messages are lost when a crash occurs. Communications between the process definition node and other nodes, for instance, can take place through non-persistent messages. The same can be said about other operations such as monitoring or auditing.

There are several aspects to the algorithms and mechanisms presented above that are crucial to the implementation of Exotica/FMQM. To provide a generic approach we have enumerated the features of the persistent message passing layer that are necessary in this implementation. Obviously, we do not expect to find all of these features in every message passing system. Therefore, in a real implementation, there are some trade-offs to be considered. For instance, IBM's MQI does not support remote GET calls and this complicates worklist management. Some other systems, like DEC's MessageQ, may provide only limited transactional semantics which would require the handling of messages in a different way to avoid data loss due to failures. Transarc's RQS, for instance, does not support non-persistent messages which does not allow one to tune performance by using non-persistent messages in cases where there is no risk of data loss. In general, each implementation will have to deal with a different set of issues depending on the particular system being used.

5 RELATED WORK

The concept of workflow management has been around for a number of years. Only recently, however, the technology required to implement suitable systems has been available. Today there are more than 70 products that claim to be workflow solutions [Frye, 1994]. Standarization efforts are underway to guarantee a common reference model across all workflow products and platforms. The Workflow Management Coalition has recently published the *Workflow Reference Model* [Hollinsworth, 1994], along with other documents, in which the basic components of a workflow management system are outlined and described.

There are many areas that have influenced or are related to workflow management. Document management, for instance, is concerned with the lifecycle of electronic documents. These systems can be traced back to early systems like TLA [Tsichritzis, 1982] where *form procedures* were proposed to organize the flow of forms. Electronic mail systems [Goldberg et al., 1992], [Malone et al., 1987], which provide means to distribute information among individuals. Office automation systems such as DOMINO, where the control and data flow in an office is controlled by a mediator that controls the execution of *office procedures* [Kreifelts and Woetzel, 1986], [Kreifelts et al., 1991]. Transaction based applications, like the model based on extended nested sagas [García-Molina et al., 1990] for coordinating multi-transaction activities, or the ConTract model, which provides reliable control flow between transactional activities [Waechter and Reuter, 1992]. Most of these early efforts were limited in their scope and did not address business processes in their full generality. More recently, several high level designs have been proposed for modeling and managing business processes [Tomlison et al., 1993], [McCarthy and Sarin, 1993], [Medina-Mora et al., 1993], [Sheth, 1994], but most of them are centralized and fail to address issues such as high availability, failure resilience or scalability. Moreover, they tend to be transactional in nature and too centered around databases, which contrast with the reference model created by software vendors [Hollinsworth, 1994]. Barbara et al. [Barbara et al., 1994] proposed INCAS for distributed workflow management. In this model, each execution of a process is associated with an *Information Carrier*, which is an object that contains all the necessary information for the execution as well as propagation of the object among the relevant processing nodes. In [Dayal et al., 1990] and [Dayal et al., 1991], a specification language and a transactional model for organizing long running activities is developed. Although intended for workflow systems, the primary emphasis is on long running activities in a transactional framework using triggers and nested transactions. The authors present a preliminary design using recoverable queues, which have similar semantics to the message queue interface (MQI) used in this paper.

6 CONCLUSIONS

Exotica/FMQM's architecture has two concrete goals. The first is to study the effects of complete decentralization on the design of a workflow product. The second is to analyze the feasibility of replacing a centralized database by persistent messaging. Both goals are of interest in workflow environments but they are broad enough to provide some insight in other areas. In particular, complete decentralization such as that achieved by Exotica/FMQM turns out to have implications that affect not only the underlying architecture but also the expressiveness of a business process, which suggests the importance of using models with enough expressiveness but no *a priori* assumptions about the system.

The gains obtained from using Exotica/FMQM's architecture are a greater resilience to failures, scalability, and the possibility of dynamically configuring the system as processes evolve. Since there is no centralized server, there are neither bottlenecks nor single points of failure. Node failures will stop the execution of process instances that use that node, but will not prevent other process instances from being executed at other nodes. The price for all this is more complex algorithms for navigation. Our conclusion is that for small or mid-sized business environments, a centralized architecture will probably suffice. However, for large or very large business environments, a distributed one provides much more flexibility and eliminates bottlenecks, thus appearing as a more promising venue for implementation.

Future work includes using two phase commit protocols to integrate legacy applications, using logs for monitoring workflow processes, and exploiting non-persistent messages to improve performance. In general, message systems are designed for messages to be consumed fast. In our system, messages may remain in a queue for a long time and indexes on message header contents may be needed to help performance. Some extensions to the message system are necessary to provide an appropriate functionality.

ACKNOWLEDGEMENTS

This work was done while Divyakant Agrawal and Amr El Abbadi were on sabbatical at IBM Almaden and Mohan Kamath was visiting IBM Almaden. 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.

REFERENCES

[Barbara et al., 1994] Barbara, D., Mehrota, S., and Rusinkiewicz, M. (1994). INCAS: A Computation Model for Dynamic Workflows in Autonomous Distributed Environments. Technical report, Matsushita Information Technology Laboratory.

[Chandy and Lamport, 1985] Chandy, K. and Lamport, L. (1985). Distributed Snapshots: determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75.

[Dayal et al., 1990] Dayal, U., Hsu, M., and Ladin, R. (1990). Organizing Long-running Activities with Triggers and Transactions. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, pages 204–214.

[Dayal et al., 1991] Dayal, U., Hsu, M., and Ladin, R. (1991). A Transaction Model for Long-running Activities. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 113–122.

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

[García-Molina et al., 1990] García-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. (1990). Coordinating Multi-transaction Activities. Technical Report CS-TR-247-90, Department of Computer Science, Princeton University.

[Goldberg et al., 1992] Goldberg, Y., Safran, M., and Shapiro, E. (1992). Active Mail - A Framework for Implementing Groupware. In *Proc. of the Conference on Computer-Supported Cooperative Work (CSCW)*, pages 281–288, Toronto, Canada.

[Hollinsworth, 1994] Hollinsworth, D. (1994). The workflow reference model. Technical Report TC00-1003, Workflow Management Coalition. Accessible via: http://www.aiai.ed.ac.uk/WfMC/.

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

[IBM, 1995a] IBM (1995a). *FlowMark - Managing Your Workflow, Version 2.1.* IBM. Document No. SH19-8243-00.

[IBM, 1995b] IBM (1995b). *FlowMark - Modeling Workflow, Version 2.1.* IBM. Document No. SH19-8241-00.

[IBM, 1995c] IBM (1995c). *FlowMark - Programming Guide, Version 2.1.* IBM. Document No. SH19-8240-00.

[IBM, 1995d] IBM (1995d). *FlowMark for OS/2: Installation and Maintenance.* IBM. Document No. SH19-8244-00.

[Kreifelts et al., 1991] Kreifelts, T., Hinrichs, E., Klein, K., Seuffert, P., and Woetzel, G. (1991). Experiences with the DOMINO Office Procedure System. In *Proceedings ECSCW '91*, pages 117–130. Amsterdam.

[Kreifelts and Woetzel, 1986] Kreifelts, T. and Woetzel, G. (1986). Distribution and Exception Handling in an Office Procedure System. In *Office Systems: Methods and Tools, Proc. IFIP WG 8.4 Work. Conf. on Methods and Tools for Office Systems*, pages 197–208. October, 22-24, Pisa, Italy.

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

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

[Malone et al., 1987] Malone, T., Grant, K., Lai, K., Rao, R., and Rosenblitt, D. (1987). Semistructured Messages Are Surprisingly Useful for Computer-Supported Coordination. *ACM Transactions on Office Information Systems*, 5(2):115–131.

[McCarthy and Sarin, 1993] McCarthy, D. and Sarin, S. (1993). Workflow and Transactions in InConcert. *Bulletin of the Technical Committee on Data Engineering*, 16(2). IEEE Computer Society.

[Medina-Mora et al., 1993] Medina-Mora, R., Wong, H., and Flores, P. (1993). Action-Workflow as the Enterprise Integration Technology. *Bulletin of the Technical Committee on Data Engineering*, 16(2). IEEE Computer Society.

[Misra, 1983] Misra, J. (1983). Detecting termination of distributed Computations Using Markers. In *ACM Proceedings of the Symposium on Principles od Distributed Computing*, pages 290–294.

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

[Sheth, 1994] Sheth, A. (1994). On Multi-system Applications and Transactional Workflows, Bellcore's projects PROMP and METEOR. Collection of papers and reports from Bellcore.

[Tomlison et al., 1993] Tomlison, C., Attie, P., Cannata, P., Meredith, G., Sheth, A., Singh, M., and Woelk, D. (1993). Workflow Support in Carnot. *Bulletin of the Technical Committee on Data Engineering*, 16(2). IEEE Computer Society.

[Tsichritzis, 1982] Tsichritzis, D. (1982). Form Management. *Communications of the ACM*, 25(7):453–478.

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

[Walk, 1993] Walk, K. (1993). Workflow Management Concepts, a White Paper. Technical report, IBM Vienna Software Development Laboratory, Austria.

BIOGRAPHY

- **Gustavo Alonso** is a Senior Research Associate in the database group of the Information Systems Institute at ETH Zürich in Switzerland. Previously, he was a Visiting Scientist at IBM Almaden Research Center, where he participated in the Exotica project working on advanced transaction models and workflow management systems. His research interests are in transaction processing, advanced database applications and distributed systems. He received an engineering degree in Telecommunications from ETSIT, Madrid Technical University, in 1989, and his MS and PhD degress in computer science from the University of California at Santa Barbara, in 1992 and 1994, respectively.
- **Divyakant Agrawal** received his Ph.D. in computer science in 1987 from State University of New York at Stony Brook. He is currently an Associate Professor of computer science at the University of California at Santa Barbara. His research interests are in databases and distributed systems and has served on several program committees of international conferences, symposia, and workshops on database and distributed systems. He has over 60 publications in refereed journals and conferences. Dr. Agrawal recently

has been involved with several large projects dealing with scientific and spatial data. As co-PI on the Amazonia project funded by NASA, Dr. Agrawal and his colleagues have developed efficient mechanisms for retrieving data scattered over heterogeneous information repositories. As co-PI on the Alexandria project (Digital Library) funded by NSF/ARPA/NASA, he in collaboration with his colleagues is involved in developing a distributed library for map and image data. His current research interests are in the area of content-based retrieval of image data and multidimensional index structures.

- **Amr El Abbadi** received his Ph.D. in Computer Science from Cornell University. In August 1987 he joined the Department of Computer Science at the University of California, Santa Barbara, where he is currently an Associate Professor. His research interests are in the fields of fault-tolerant distributed systems and databases. He has designed several protocols for highly available fault-tolerant systems. He has also been involved in designing systems and database support for geographic information systems and collaborative environments. Recently, he has been involved in The Alexandria Digital Library project, whose goal is to design a distributed spatial digital library. He was ACM lecturer for 1991-92, and is currently the area editor for Information Systems: An International Journal.
- **C. Mohan** is a Research Staff Member at IBM's Almaden Research Center since 1981 and is a member of the IBM Academy of Technology. He is currently leading the Exotica project on workflow systems. He is a designer and an implementor of R*, Starburst and DB2/6000. He is the primary inventor of the ARIES family of locking and recovery algorithms, and the industry-standard Presumed Abort commit protocol. His research ideas are incorporated in numerous products (e.g., DB2 Family, S/390 Parallel Sysplex Coupling Facility, MQSeries, ADSM). Mohan has received 5 IBM Outstanding Innovation Awards, and is an inventor on 14 issued and 13 pending patents. He was the Program Chair of the 1987 International Workshop on High Performance Transaction Systems and a Program Vice-Chair of the 1994 International Conference on Data Engineering. He will be the Americas Program Chair for the 1996 International Conference on Very Large Data Bases. He is an editor of VLDB Journal. Mohan has a PhD from University of Texas at Austin and a BTech from IIT Madras.
- **Roger Günthör** received a Dipl. Inform. in computer science from University of Stuttgart in 1990. Afterwards, he became a Ph.D. candidate at the IPVR at the University of Stuttgart. His thesis about reliable base services for long-lived activities was written in the context of the ConTracts project. In August 1994, he joined the Exotica group at the IBM Almaden Research Center. His areas of interest include transaction processing, extended transaction models, workflow management, scheduling, temporal logic, and client/server technologies.
- **Mohan Kamath** is a PhD candidate in the Department of Computer Science at the University of Massachusetts, Amherst. After receiving his undergraduate degree in engineering from India, he received a Master's degree in Computer Integrated Manufacturing and Design from the University of Maryland, College Park and a Master's degree in Computer Science from the University of Massachusetts, Amherst. His research interests include transaction processing, workflow management, mobile computing and multimedia database systems.