Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System*

G. Alonso R. Günthör M. Kamath† D. Agrawal‡ A. El Abbadi‡ C. Mohan
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120, USA
{gustavoa, rgunther, mohan}@almaden.ibm.com, {agrawal, amr}@cs.ucsb.edu, {kamath}@cs.umass.edu
Research

Abstract

Workflow Management Systems (WFMS) are a first generation of products that attempt to manage the execution of business processes by large numbers of users distributed over a wide area and using heterogeneous resources. They are a very promising venue for collaborative systems but, in most cases, the autonomy of the users is greatly restricted due to architectural and design considerations. This is a severe restriction, especially when considering the emergence of mobile computing, and the increase in use of laptops and small computers which are connected to a network only occasionally. In this paper, we discuss how disconnected workflow clients can be supported while preserving the correctness of the overall execution and allowing coordinated interactions between the different users. Disconnected client support provides a great deal of flexibility in the usability of a WFMS and enhances the system's resilience to failures.

1 Introduction

The recent increase in the availability of powerful desktop computers and high-speed networks has allowed organizations to move towards distributed computing environments. Such environments allow organizations to coordinate their staff and resources that are distributed across a wide geographic area. As part of this trend, *Workflow Management Systems*, WFMSs, have recently attracted much attention for their ability to improve the efficiency of an organization by streamlining and automating business processes. A WFMS supports the modeling, coordinated execution and monitoring of the activities that take place within an organization. In general, WFMSs are collaborative tools that allow the administrator to define the structure of the business processes and to assign the staff and role hierarchies in the organization within which those business processes will be executed. Users within an organization cooperate effectively in the completion of business processes by letting the WFMS coordinate their activities. Furthermore, users only need to focus on the task at hand. The routing of the flow of control and data, the location of resources and the assignment of personnel is done automatically by the system. Hence, in a workflow environment, cooperative work is defined in terms of business processes whose execution is controlled by a workflow management system. Users collaborate with each other by executing individual steps within a business processes. The synergy between all these steps is provided by the designer of the business process. Once the process has been defined, its actual coordinated execution is ensured by the system : the WFMS delivers the task to perform to each user, collects results, determines the next steps, controls the activities of each user, and detects when the process has been successfully terminated. Note that a workflow management system is not fully automated, since human intervention is necessary to solve many crucial steps and to determine what to do in case of errors and unpredictable events. However, the use of a workflow management system simplifies to a great extent the task of coordinating large numbers of users working on heterogeneous and distributed environments.

Most WFMSs are based on a client-server architecture [AKA+94] in which, though individual steps may be executed at geographically distributed nodes, the knowledge about the processes being executed is kept in a centralized database at the server. This centralization makes it easier to synchronize and monitor the overall execution. Unfortunately, this forces the clients to be connected to the server at all times to be able to

*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.

†Computer Science Department at the University of Massachusetts, Amherst, MA 01003, USA.

‡Computer Science Department, University of California at Santa Barbara, CA 93106, USA.

progress in their work. In recent years, there has been a growing trend towards the use of portable computers and home desktop computers for office work. Users load applications and data in their laptops or desktops by briefly connecting with a server in the office. Then they disconnect from the server and work on those applications and data. After the work has been completed, which may be in a few hours or few days, they reconnect with the server and store the results of their work, maybe downloading more data and application to repeat the process again. This mode of operation has been widely acknowledged as one of the main ways in which computers will be used in the future [IB94]. Note that this includes not only mobile terminals but also desktops or any other computer type connected to the server only occasionally via a modem. Disconnected operation greatly expands the scope of an organization's distributed computing infrastructure with an obvious benefit. However, since users are not permanently connected to the system, it becomes more difficult to coordinate their work.

The main issue we address in this paper is how to support disconnected computing within WFMSs. Note that, to a certain extent, disconnected computing and workflow management systems have contradictory goals. The latter is a tool for cooperation and collaborative work in which users work within a preestablished framework that guarantees progress towards a certain goal, the business process, of which the users may not be aware. This requires constant monitoring and checking of the users activities. On the other hand, disconnected computing is targeted to users who work in isolation from other users. There is not much room for collaboration in disconnected mode. The solution we propose is a compromise between both worlds. The users who want to work in disconnected mode, must "commit" themselves to perform certain tasks. The workflow management system takes advantage of this guarantee to associate tasks with users, allowing them to work on their own while ensuring overall correctness and constant progress towards the goal of the business process. To our knowledge, this is the first study of the impact of portable computers on collaborative workflow systems, and to provide a feasible solution where the implementation aspects are discussed within the constraints of a real system: FlowMark [IBMb, IBMa, IBMc], a workflow product from IBM that uses a client-server architecture. As stated in [IB94], there is a tremendous potential in combining these technologies since it could provide an entire new outlook to cooperative workflow systems.

The rest of the paper is organized as follows: Section 2 briefly introduces FlowMark and discusses its client-server architecture. In section 3 we describe the specifics of our approach to integrate a disconnected client into FlowMark. Section 4 discusses related work, and Section 5 concludes the paper.

2 FlowMark and Workflow Systems Architecture

Workflow is, in general, an ill-defined concept [Rei93, Hsu93], which makes it difficult to provide a general description of its characteristics and goals. Hence, for clarity and to motivate the rest of the paper, we focus our attention on a specific workflow model with enough features in common with other approaches to be considered a general framework. This is the model of FlowMark [LA94, LR94], an IBM workflow product designed for managing business processes in distributed and heterogeneous environments.

2.1 Business Processes

A business process coordinates the different steps required to achieve a particular goal [Hol94]. Its key elements are thus the steps to execute and how they are coordinated into a meaningful whole. Consider, as an example, the approval of a loan request in a bank. The first step involves getting information from the person requesting the loan. Then, and only after the first step has been completed, a credit report is obtained, either by consulting the bank records or from an outside agency. This information is then used to make a case study, comparing with other cases, using information about past loans to the same customer, and analyzing the market conditions. Once a recommendation is ready, a decision must be made. If the loan is rejected, a letter is sent to the customer. Otherwise, the appropriate documents are prepared, the customer's signature is obtained and the loan is finally given. This business process is shown in Figure 1.

Each separate step or *activity* in the business process has a goal of its own. It is, however, the coordinated execution of all of them that makes the business process. Some of the steps may even be as complex as entire business processes themselves. For instance, in Figure 1 *case study* and *finalizing the credit* are shown as nested processes. In practice, users define processes in terms of other processes and, hence, the notions of nesting and modularity are crucial to understand

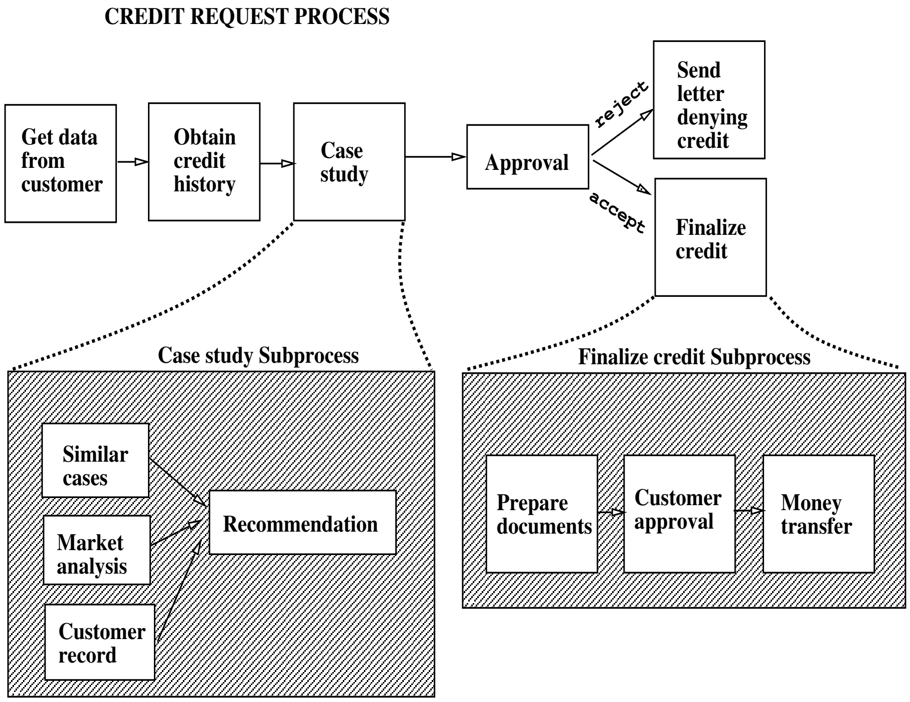

Figure 1: A Loan Request as an Example of a Business Process

business processes. But business processes are just modeling tools that express sequences of events within an organization and provide them with some higher level interpretation. The actual execution of such business processes is performed by the WFMS. In general, a WFMS provides a modeling tool to represent business processes and a runtime system to coordinate and execute them in a heterogeneous and distributed environment. In addition to this, WFMSs provide special features that are essential to the implementation of real business processes and that differentiate them from other systems like decision support tools, databases, transaction monitors, or mail tools. The details of a WFMS are beyond the scope of this paper, but to provide some insight into their expressiveness consider the following. The most characteristic feature in a WFMS is the ability to define the organization within which the business processes will take place. This includes different roles such as managers, directors, and programmers, for instance; hierarchical levels such as manager-1, manager-2, or programmer-3; and different types of staff, i.e., the final users of the system. The mapping between users, levels and roles is many to many. This provides a great deal of flexibility when defining and executing a process. During the execution of a process, the WFMS determines who is responsible for carrying out a certain task. This can be done by associating each step with a role such as "programmer", for instance. All users who fit that description become eligible to execute that step and are notified of the fact that the step is ready for execution. This may be used to perform some load balancing among the users and, in the absence of a particular user, the progress of a process will not be interrupted: a step associated with several users can be executed by anyone of them. Moreover, it is also possible to specify who must be notified if the step is not executed within a certain period of time. All of these features correspond to real concepts in a work environment and are crucial to the success of a WFMS.

2.2 Model

Business processes are modeled in FlowMark as acyclic directed graphs in which nodes represent steps of execution and edges represent the flow of control and data among the different steps [IBMa, IBMb]. As shown in Figure 2, the main components of a FlowMark *workflow model* are: *processes*, *activities*, *control connectors*, *data connectors*, and *conditions*. A process is a description of the sequence of steps involved in accomplishing a given goal and it is represented as a graph. Activities are the nodes in the process graph and represent the steps to be completed. Control connectors are used to specify the order of execution between

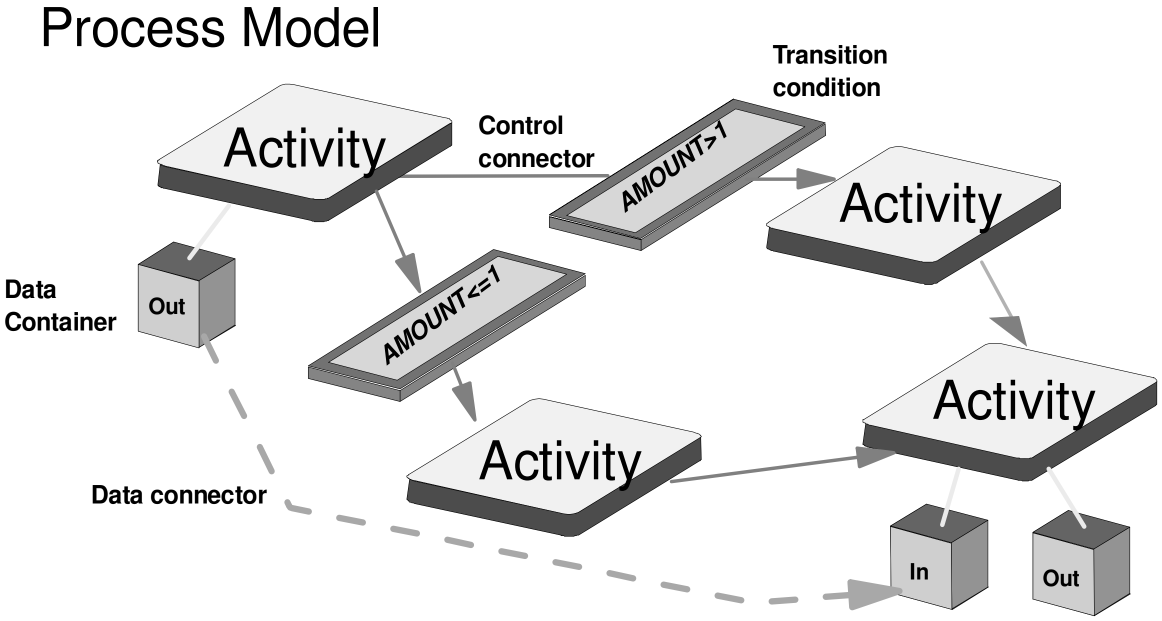

Figure 2: *FlowMark Representation of a Business Process*

activities and are represented as directed edges in the process graph. Data connectors specify the flow of information from one activity to another and are also expressed as directed edges in the process graph. Finally, conditions specify when certain events will happen. There are three types of conditions: *transition conditions*, associated with control connectors and specify whether the connector evaluates to true or false; *start conditions*, which specify when an activity will be started; and *exit conditions*, used to specify when an activity is considered to have terminated.

For our purposes here, the most relevant aspect of the model is the use of *worklists*. A worklist is a list of workitems associated with a user. Each workitem is an activity that belongs to a business process being executed and that has been assigned to this user, and possibly also to others, for its completion. Each user has a worklist, and activities may appear in several worklists at the same time. When an activity is selected in one worklist, it will be deleted from all other worklists. Worklists can be seen as the interface of the WFMS to the end user. Ideally, users are not aware of the processes that trigger an activity. They do not have to worry about the flow of control or data. They do not even need to know which other users are participating in the execution of the process. Users only see their worklist, where the activities they have to execute are displayed by the WFMS.

From the worklist point of view, activities are the key items. A process specifies the order in which a sequence of activities is to be executed. *Data containers* specify the relevant data for each activity. Conditions specify when activities are actually allowed to be executed. All of these tasks are performed by the WFMS. Every time an activity is eligible for execution, it is broadcast to the worklists of the users associated with that activity. When a user selects an activity from its worklist, the activity will be deleted from all other worklists and become associated with that user. Once the execution of the activity is completed, the activity is deleted from the worklist. As a result of its execution, new activities may become eligible for execution and the cycle begins anew.

2.3 Architecture

The current version of FlowMark is centered around an object oriented database, ObjectStore. FlowMark runs across different platforms - AIX, OS/2, Windows - and its components can be distributed across heterogeneous systems. The centralized database is used to store all the information about the schema and runtime aspects of processes. All other components work by sending messages reporting the occurrence of an event, and getting messages instructing them about what to do next. Communications are not directly with the database but conducted through a server, the *FlowMark Server*. The other components of FlowMark are: *Runtime Client*, *Program Execution Client*, and *Buildtime Client*. Each of these clients may reside in a different host. The logical connections among the runtime components are shown in Figure

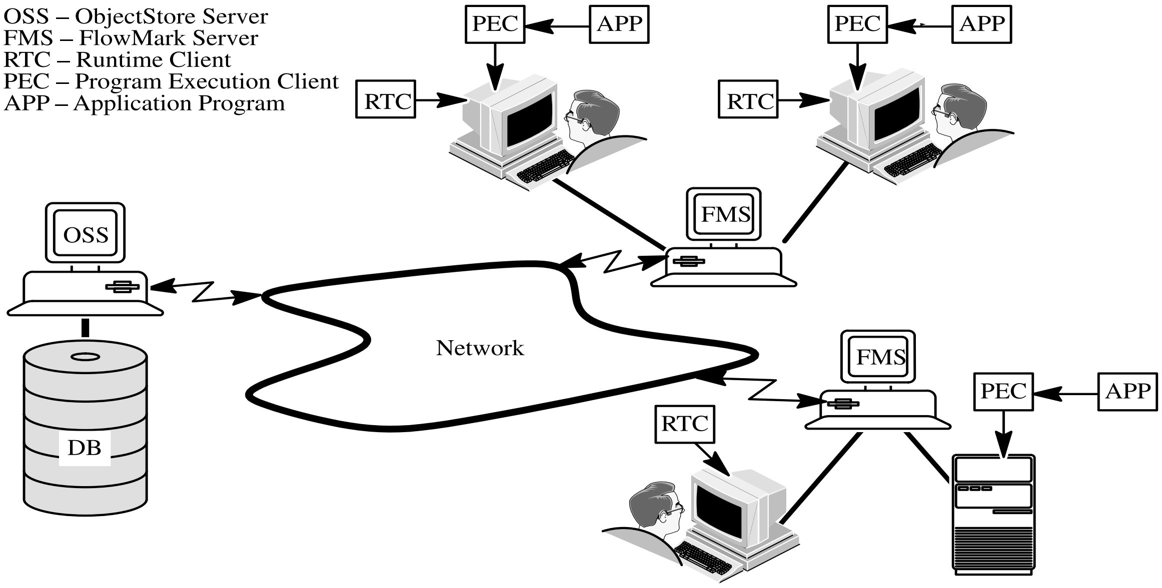

Figure 3: *Current Architecture of the Runtime Components of FlowMark*

3. The workflow servers are connected to the centralized database in a client/server fashion. Connected to each workflow server there can be several local or remote runtime clients and program execution clients. Runtime clients can be connected to several servers simultaneously. Communication among these components takes place through TCP/IP, NetBIOS or APPC, except for the database and the workflow servers that function as ObjectStore client/servers and use their own internal protocol.

FlowMark servers relay messages between the database and all other components. The definition and creation of workflow models and all other related information such as staff definition, and role assignments, takes place in the buildtime clients, which interact directly with the database server. Runtime clients are the interface to the end users and manage the users' worklists. Program execution clients control the actual execution of the activities. They are the interface between FlowMark and the external application environments where the programs run. They spawn the programs, monitor their termination and return the appropriate output container variables to the runtime server.

There is no restriction on the nature of the activities included within a process as long as there is a way for FlowMark to trigger their execution. Note that they do not necessarily need to be computer programs. Phone calls, meeting and many other human activities can be included as part of a process as long as there is an interface to trigger their occurrence and record their results upon termination. The required coordination between activities is performed by FlowMark by scheduling them in the order specified in the process definition. If an activity is set to be executed after another activity, FlowMark will wait until the successful completion of the earlier activity before setting the later activity as being ready for execution. FlowMark will also take care of execution paths that are never reached because of the conditions that have been set.

When an activity becomes ready for execution, FlowMark performs role and staff resolution to determine all the users who are eligible to execute that activity. It then updates the worklists of all these users by including the activity as a new workitem. This involves sending an *activity ready* message to the runtime clients where those users are logged on.

The exchange of messages that occurs among the different components when an activity is executed, shown in Figure 4, is as follows: first, the user indicates that an activity must start. This is done by selecting the activity from the user's worklist, which results in the runtime client sending a *start activity* message to the server. Upon receiving such a message, the workflow server initiates a transaction on the database to determine the state of the activity and where it should be executed. Once the transaction commits, the server sends a *start program* message to the corresponding program execution client. The server also sends an *activity running* message to the runtime client that will result in the status of the activity being changed in the worklist. Finally, the server notifies

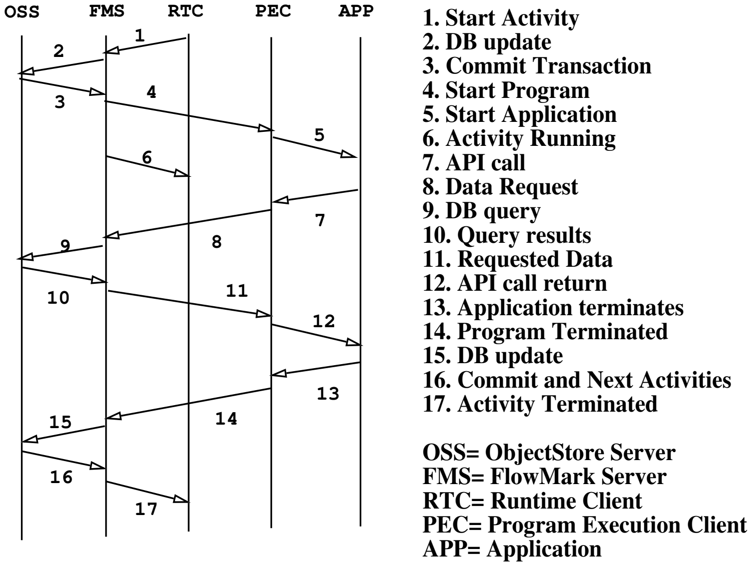

Figure 4: *Message Exchange Between the Different FlowMark Components During the Execution of an Activity*

all other users who were eligible to execute the activity that the activity is already executing. The messages sent to the corresponding runtime clients for this purpose result in the activity being deleted from those clients' worklists.

Once the program execution client receives a *start program* message it starts the application. The application may request information from its input container through APIs. These are received by the program execution client and, if the information is locally available, it will relay it to the application. If the information is not locally available, it will request it from the server and then relay it to the application. An application may not make use of the API calls, so this round of messages is optional. The program execution client captures the termination of the application and sends a *program terminated* message to the server with any values the application returned. At the server, this message triggers the execution of a transaction that will store the values returned by the application in the appropriate output data container, and check the exit condition of the activity. If the activity terminates successfully, then the same transaction determines the next activities to execute. When this transaction commits, the server sends *activity ready* messages to all the runtime clients where the users eligible to execute those activities are logged on. It also sends an *activity terminated* message to the original runtime client indicating that the activity completed its execution. Note that the entire sequence of messages is organized based on the fact that all components are always connected to the server and, therefore, to the database. The clients lack decision making capabilities and the information necessary to take steps on their own. Everything is done in lockstep with the updates made to the database, which simplifies synchronization and the design of the clients. This synchronicity between server and clients is what needs to be changed to support disconnected clients.

3 Supporting Disconnected Clients

In this section we discuss the requirements of the system to support disconnected clients in terms of changes to actual components. Note that we intend to only modify the clients, leaving the workflow server and the database schema intact. The basic idea is to provide the clients with enough information to allow them to proceed without having to consult with the server after every step. There are two possibilities, one is to have the clients working in a "batch" mode, where a set of activities is assigned to them and all the relevant information is downloaded to the clients prior to their disconnection. The other is to allow the clients to perform navigation themselves by transferring entire parts of a process to the clients. For reasons of space, we will explore only the first option in detail. The possibility of performing navigation at the clients will be briefly discussed at the end of this section. For simplicity in the exposition, we will assume FIFO communications.

3.1 System Requirements

As illustrated by the previous description of the exchange of messages that occurs when an activity is being executed, FlowMark operates by maintaining all the information in a centralized database. All other components, workflow servers and clients, have no persistent memory and have no information regarding the actual processes being executed beyond that required to execute activities. The FlowMark server makes all the decisions, which are communicated to the clients. The clients only act in a reactive manner to messages from the server.

To allow clients to work while disconnected, they must have more autonomy and their own storage. They should also have access to the information necessary to be able to proceed without consulting the centralized database. Note that in the current version of FlowMark, a program execution client may be disconnected after receiving a *start program* message. In spite of being disconnected, it will still execute the application, capture its termination and wait until it is connected back to the server to send the results of the execution. However, when this type of disconnection occurs, the program execution clients can not proceed to execute any other activities.

The fact that the clients are more autonomous implies that the server must ensure there are no conflicts between the different disconnected clients by not allowing two users to work simultaneously on the same activities. Thus we will assume that the clients will notify their intention to work in disconnected mode to allow the server to take the necessary steps to avoid incorrect execution or conflicting results. This is not a very restrictive assumption since such intention can be easily triggered by the user when declaring that some work is to be loaded in the clients. This is a common procedure in laptops and remote terminals that are connected to the network only occasionally.

During disconnected operation both the runtime client and the program execution client are necessary or, at least, their functionality. The user still interacts with the system through worklists, managed by the runtime client, while the actual execution of applications is handled by the program execution client. During disconnection both clients reside in the same machine, so they could be combined in a single component. However, to avoid unnecessary changes, we keep them as separate components. These clients need to be modified to handle the fact that the server is not accessible during disconnection.

Runtime clients in FlowMark play a passive role. They are mere interfaces for the user to specify actions such as *start activity*. As such, the role of a runtime client does not change in disconnected mode except for the fact that instead of sending the commands to the centralized database, in disconnected mode it will send them to the program execution client. Similarly, the execution progress of an activity will not be reported by the database but by the program execution client. Thus, for all practical purposes, the runtime client does not change, except for some added functionality that will be discussed later.

During normal operation, the program execution client acts upon the messages sent from the server and works as a relay between the application and the server. During disconnected operation, the program execution client will act according to the messages received from the runtime client. However, during disconnected operation, the program execution client cannot connect to the database to provide additional information requested by the application through API calls. Thus, it must provide its own persistent storage where the information that maybe requested by the application is stored in advance. Similarly, it must also store the results of the application's execution until they can be sent to the server.

The sequence of messages exchanged during disconnected operation between the different components is summarized in Figure 5. Note that this involves a previous phase in which the clients and the servers "agree" on the disconnection, or *planned* disconnection in the mobile computing literature, a phase in which the clients are disconnected, and a phase in which the clients are reconnected to the server. In what follows, the different phases will be discussed in detail.

3.2 Synchronization Prior to Disconnection

This preliminary phase involves two important steps: locking and loading the activities that will be available at the clients during disconnected mode.

Locking is necessary due to the fact that activities may appear in several worklists simultaneously. Under normal circumstances, the centralized database serializes all changes to an activity and, hence, even if two users start the same activity concurrently, only one of them will be able to register in the database as the user to which the activity has been assigned. All other requests arriving later will be rejected. In disconnected mode, the system must ensure that disconnected clients do not work simultaneously on the same activity. Therefore, for disconnected operation, users must declare

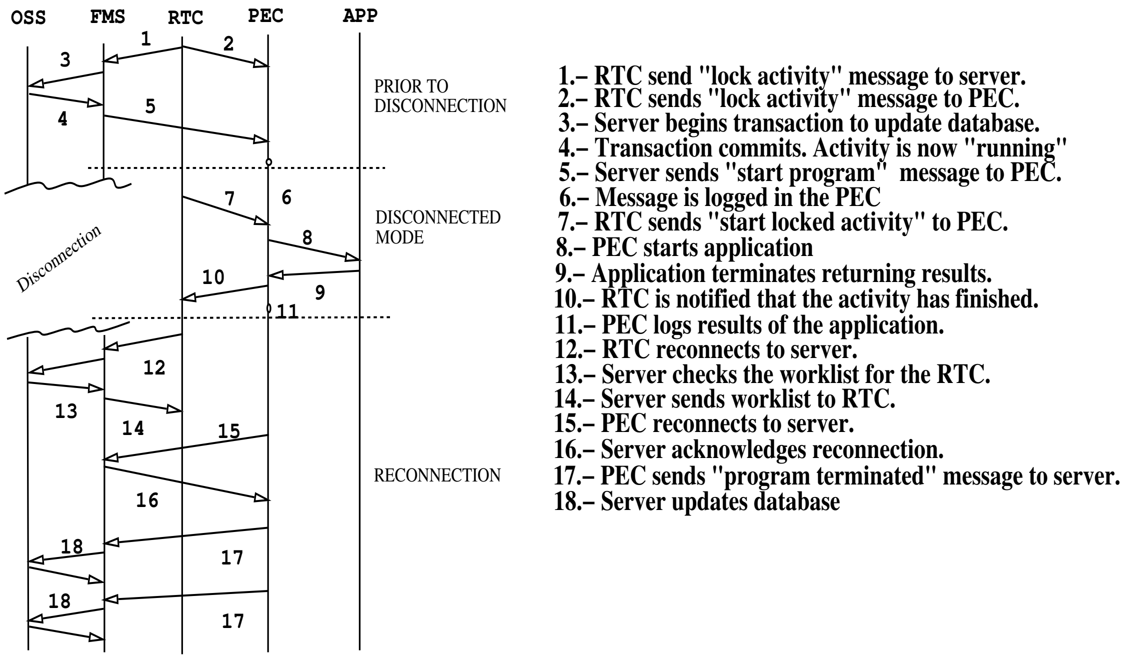

Figure 5: *Message Exchange Between the Different FlowMark Components During Disconnected Operation*

their intention to work on a particular activity. Before a user can disconnect from the server, the user must *lock* the activities to work with. Locking takes place at the runtime clients and although a locking operation is similar to starting the activity from the server point of view, the runtime and program execution clients require some modifications. Since a *locked activity* is not to be executed until later, after the clients have been disconnected from the server, such a state must also be defined in the clients. A locked state for an activity implies that the user will eventually perform the activity and that it will be executed in disconnected mode. An activity in the worklist can enter the locked state only from the ready state: activities already running cannot be locked. From the locked state an activity can be started - and then it goes into the running state - or manually finished - in which case it goes into the ready state. In the program execution client, it must be possible to detect that the data that is arriving from the server corresponds to a locked activity and not to an activity to be executed immediately. To ensure that the program execution client can differentiate between normal activities and locked activities, some form of hand-shaking is necessary between the runtime client and the program execution client. This data must be stored and properly indexed for later retrieval.

When a user locks an activity, a *lock activity* message is sent to both the server and the program execution client. In this way the program execution client knows the difference between a locked activity and other type of activities: upon receiving such a message, it will store the activity in a list of locked activities. When a *lock activity* message arrives, the server behaves as if it were a *start activity* message; it sends a *start program* message to the corresponding program execution client. The program execution client compares all incoming *start program* messages with its list of locked activities. If the application to be executed corresponds to a locked activity, instead of starting the application, it will store the message for later use. Then, the program execution client requests the input data container for that activity from the server, and upon receiving it, stores it for future use. In this way, the program execution client will be able to answer APIs from the application. Once all this is done, the program execution client notifies the runtime client, which then considers the activity to be successfully locked. Before disconnecting, the program execution client should check whether the applications corresponding to the activities that have been locked are present. If they are not, an error message is sent to the runtime client. The activity is not immediately discarded since the user may install the application after the messages have been sent. The error message is a warning. Once disconnected, the clients will not be able to find another program execution client where the activity can be executed.

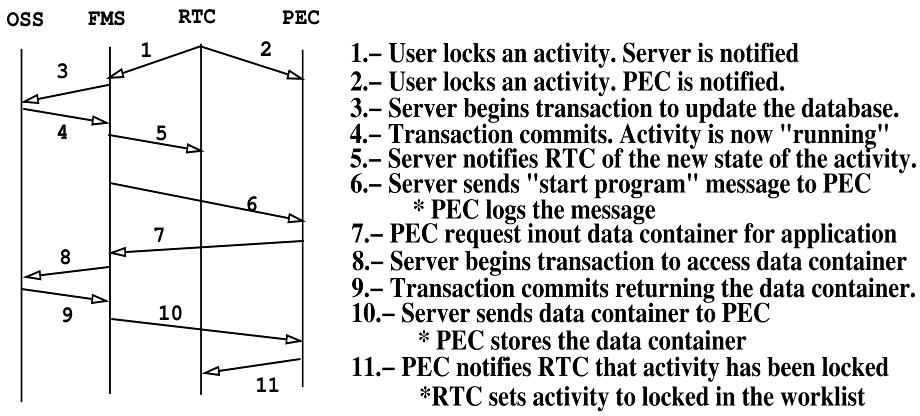

Figure 6: Messages Exchanged Prior to Disconnecting from the Server

Conflicts on the requests of locks on activities are solved through the serialization that takes place in the workflow server and database. If a request for a lock arrives too late, it is simply discarded. The exchange of messages involved is summarized in figure 6. This exchange is a refinement of that shown in Figure 5.

Note that locking an activity represents a commitment on the part of the user to perform the activity. From the workflow server point of view, the activity has already started and it is being executed. If the user does not perform the activity, it will eventually time-out and appropriate action will be taken by the workflow server. This may involve notifying somebody of the situation, or deleting the activity from the worklist of the user and sending it to the worklist of another user. It is possible to address this issue in several ways. One is to assume that the user will perform the activity. Another is to force finish the activities that have not been executed and notify the server, which will unlock the activity and then resend the activity to other worklists. Both of these options can be implemented and have no effect on the overall design, so we will not discuss them further.

Finally, note that it is possible for activities to be running at the local site at the time the user wants to disconnect it from the server. Since these activities are known to the runtime client, and the disconnection operation is also processed by the runtime client, the system is able to detect these cases and delay the disconnection until there are no activities running at that site.

3.3 Disconnected Operation

During disconnected operation, a user can start only locked activities. This can be enforced by removing from the worklist all activities that are not locked at the moment of disconnection. Upon starting a locked activity, the runtime client sends a *start locked activity* message to the program execution client. Upon receiving this message, the program execution client retrieves the data from its own repository and proceeds as usual when executing an application. Since the input data container of the activity is available at the program execution client at the moment execution starts, APIs to request data from this container can be served from the program execution client without accessing the database.

When the execution of an application terminates, the values it returns are captured by the program execution client. Under normal circumstances, the program execution client would send this data to the server immediately. During disconnected operation, however, this is not possible and, hence, it must store the results until it is reconnected to the server. Note that there is a chance for the data to get lost if the program execution client fails before storing the results in stable storage. This problem also occurs during normal operation and it is beyond the scope of this paper. The termination of the application is notified to the runtime client, which will remove the locked activity from the worklist. However, the runtime client still keeps the name of the activity in a list of locked activities. The reason for this will become clear when discussing the reconnection phase.

The program execution client can store the messages that will be sent to the server in a sequential manner. No index is necessary since this information will only be accessed once, and the order in which the messages are sent does not matter.

3.4 Reconnecting to the Server

Reconnection to the server is straightforward. Both clients will reconnect as if they were new.

The runtime client discards, upon reconnection, all the information it has in its worklists and gets a new worklist from the server. The program execution client waits for the reconnection to finish and then starts sending the *program terminated* messages it has stored. The runtime client possibly needs to update the worklist from the server before actually displaying it since some of the activities reported by the server may have been executed during the disconnection. For this purpose, the runtime client keeps a list of the activities it had during disconnection and matches it with the new list sent by the server, doing the appropriate changes. Activities that have been executed will be deleted from the worklist and, once the program execution client sends the corresponding message, the server will learn about it. This is why it is necessary to keep a record of locked activities that have been executed while disconnected. Activities that were locked but not executed will remain locked. The program execution client keeps the information required to execute the activity until the activity is actually executed, independently of how many times the clients are connected and disconnected from the server.

3.5 Locking Sets of Activities

We are currently studying the possibility of locking blocks of activities rather than individual activities. This will allow the user to download and execute a series of related activities. Consider, for example, a credit application containing a block of activities to review a loan request: for loans greater than $10,000 a manager must review the loan request form, and either fill out a credit acceptance form or edit a letter to reject the loan request. Then the acceptance form or the reject letter is sent back to the clerk who works on the customer to take further actions or to inform the customer by mailing the reject letter. This is a sequence of activities to be performed by the same user. However, locking a number of loan requests on the worklist will not suffice. The problem is that activities not ready to be executed do not appear on any worklist. The manager would have to lock the whole block of related activities.

If only activities appearing in the worklist can be locked, then it is not possible to achieve this kind of semantics within FlowMark. To provide all the navigation features of FlowMark, e.g. starting sub-processes, on the client's side would result in installing a full-fledged FlowMark server and replicating the templates for the transitive closure of all potentially called sub-processes on the manager's laptop. Also, locking activities that are not yet ready to be started may cause problems. Consider, for example, an activity waiting for control and data arriving from an activity executed by user A. If user B wants to lock this activity, it cannot be downloaded to its machine, since the data containers have not been filled up yet. In what follows, we will not consider such a case.

Instead, we introduce a new concept: *simple blocks*. A simple block may contain elementary activities, simple blocks, and blocks for which the same holds, only. It must not contain subprocesses. A simple block is different from a normal block, in that it explicitly shows up on a worklist. When a user selects a simple block from the worklist, all its activities are assigned to this user. Any users or roles assigned to the activities at build-time are overruled. Thus, all the activities recursively contained in a simple block will be executed by the same user. If the user locks a simple block, all the activities in the block are locked by this user and the user is committed to execute each of them. Note however that only the activities ready for execution actually appear on the user's worklist so the runtime client would have to keep track of them on a separate data structure. (Alternatively, one could think of displaying these activities on the user's worklist, too, but graying them out to show that they are not ready for execution, yet.)

Locking simple blocks requires changes to various FlowMark components, including both the FlowMark build-time client and run-time server. For instance, simple blocks must be checked to ensure that they do not call subprocesses, activities in a simple block must be executable on the same machine, a simple block must be assigned to a role, automatic execution is not allowed, and the program execution client must be able to deal with all the information about activities, containers and navigation, effectively performing the same task as the server and the database.

4 Related Work

There is a considerable amount of reserach in the area of workflow [She94, Hsu93, GHM+93, TAC+93, WR92, DHL91], however, both in research and in commercial systems, the issue of disconnected client support has not been fully addressed. Workflow systems like FlowMark, InConcert [MS93], or WorkFlo [FG87] assume users to be permanently connected to a more or less centralized workflow server while working on their worklist.

Recently, several proposals for distributed workflow management architectures have been made that address the issue of disconnected and unavailable clients. In Exotica/FMQM [AAE+94] and in the architecture discussed for the INCA model [BMR94], there is no central node that controls the execution of a workflow process. Workflow control and the current status of workflows are distributed across a number processing nodes, which are considered to be disconnected or unavailable quite frequently. Both systems use a reliable store and forward messaging mechanism to transfer activities to the next processing node, thereby avoiding the need for permanent connections. Hence, disconnections only result in a delay of the messages to be send.

These systems do not incorporate a crucial concept of workflow management, i.e. to offer an activity to multiple users that are eligible to execute it. Since role resolution is done as soon as an activity becomes ready for execution, it is used just like an indirect addressing mechanism. Instead of putting the activity on a shared worklist or putting it on the individual worklists of multiple users like FlowMark does it, the activity is send to the worklist of a specific user. Hence, user flexibility to share the workload is lost.

Bussler [Bus95] has recently proposed a system to support mobile clients in a workflow environment. This system requires the workflow designer to specify whether an activity can be executed in disconnected mode. It is based on the idea of downloading containers to the client, which ensure that the execution step is self contained. These containers also include the application to be invoked, that must be installed in the client by the WFMS before disconnection occurs.

In addition to the above research projects, there are currently more than 70 products that claim to be workflow solutions [Fry94]. Many of these systems use store and forward messaging, since they are based on Lotus Notes or on e-mail. Some of them might claim themselves to support disconnected clients, however, they do not provide the full functionality of a real workflow system. These systems either do not support roles at all or role resolution is already done at the time the next activity is ready to send. As a consequence, each activity appears on the worklist of a single user, only.

In more elaborated systems like X-Workflow from Olivetti [Oli94], a central mediator controls the control and data flow of a business process by distributing semi-structured e-mail messages to all users eligible to execute an activity. Once, a user selects an activity, an execution request is send to the mediator, which assigns the activity to the user and sends him the context data (input parameters) needed to execute the activity. Also, the mediator informs other users of this fact by sending a message, which causes the intelligent mail system to remove the activity from the user's mail folder. In such environments, disconnected clients may still work on the last activity that they requested and received. However, support for downloading a set of activities is not provided, although our approach could easily be adopted in this system.

5 Discussion

The importance of disconnected operation in workflow environments has been pointed out elsewhere [IB94], however little has been done until now to analyze and determine its requirements and limitations. We have proposed a design for disconnected clients that is based on an existing workflow management system and that takes into account the collaborative aspects of the technology. While workflow systems are tools for cooperation, portable computers are generally viewed as tools for individual work. This is reflected in the architecture adopted by most workflow management systems that tends to be heavily centralized and requires the users to be connected to the central server. We have proposed in this paper a series of mechanisms that allow a workflow management system to support disconnected clients with minimal changes to the semantics of a business process and its implementation, effectively allowing users distributed over a wide geographic area and working with heterogeneous resources to cooperate, while preserving their mobility and independence.

Most of the ideas discussed above are borrowed from several areas. Our contribution is to point out the impact of disconnected clients on workflow management, which - to our knowledge - has been overlooked by previous research. In particular, we define the semantics of loading work to a mobile, disconnectable, computer by introducing the notion of *locked activities* and the user's commitment to eventually execute them. Note that locked activities stay within the user's machine and that we allow the user to execute these activities even when the connection has been re-established. Therefore, locked activities also provide a way for users to select their preferred activities remove them from the worklists of other users or to manually cache the workflow context for a number of activities at times of low network traffic. Finally, we prove the feasibility of our concept by outlining the implementation issues for FlowMark, an existing workflow management system. Note that the approach taken requires no changes to the basic workflow server and to FlowMark's workflow model. The only components affected are the clients.

Future work includes extending the mechanisms for loading blocks. This will help in migrating parts of an process instance from one FlowMark server to another. It can also be used to replicate process instances on a remote FlowMark server, thereby providing more flexibility for distributing the overall workload and to enhance fault-tolerance.

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.

[BMR94] D. Barbara, S. Mehrota, and M. Rusinkiewicz. *INCAS: A Computation Model for Dynamic Workflows in Autonomous Distributed Environments.* Technical Report, Matsushita Information Technology Laboratory, April 1994.

[Bus95] C. Bussler. *User Mobility in Workflow-Management-Systems.* In **Proceedings of the Telecommunications Information Networking Conference (TINA '95)**, Melbourne, Australia, February 1995.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. *A Transaction Model for Long-running Activities.* In **Proceedings of the Sixteenth International Conference on Very Large Databases**, pages 113–122, August 1991.

[FG87] W. Fisher and J. Gilbert. *FileNet: A Distributed System Supporting WorkFlo; a Flexible Office Procedures Control Language.* In **IEEE Computer Society Office Automation Symposium**, pages 247–249, Gaithersburg, MD, April 1987.

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

[GHM+93] D. Georgakopoulos, M. F. Hornick, F. Manola, M. Brodie, S. Heiler, F. Nayeri, and B. Hurwitz. *An Extended Transaction Environment for Workflows in Distributed Object Computing.* **IEEE Computer Society Bulletin of the Technical Committee on Data Engineering**, 16(2):24–27, June 1993.

[Hol94] D. Hollinsworth. *The Workflow Reference Model.* Technical Report TC00-1003, Workflow Management Coalition, December 1994.

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

[IB94] T. Imielinski and B. R. Badrinath. *Mobile Wireless Computing: Solutions and Challenges in Data Management.* **Communications of the ACM**, 37(10), October 1994.

[IBMa] IBM. **FlowMark - Managing Your Workflow, Version 2.1.** Document No. SH19-8243-00, March 1995.

[IBMb] IBM. **FlowMark - Modeling Workflow, Version 2.1.** Document No. SH19-8241-00, March 1995.

[IBMc] IBM. **FlowMark - Programming Guide, Version 2.1.** Document No. SH19-8240-00, March 1995.

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

[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.

[MS93] D. R. McCarthy and S. Sarin. *Workflow and Transactions in InConcert.* **IEEE Bulletin of the Technical Committee on Data Engineering**, 16(2):53–56, June 1993.

[Oli94] Olivetti Systems & Networks GmbH. **Ibisys X_Workflow-Vorgangssteuerung auf der Basis von X.400**, 1994. Produktbeschreibung.

[Rei93] B. Reinwald. **Workflow Management in verteilten Systemen.** TEUBNER-TEXTE ZUR INFORMATIK. B.G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig, 1993.

[She94] A. Sheth. *On Multi-system Applications and Transactional Workflows*, 1994. Collection of papers from Bellcore.

[TAC+93] C. Tomlison, P. Attie, P. Cannata, G. Meredith, A. Sheth, M. Singh, and D. Woelk. *Workflow Support in Carnot.* **Bulletin of the Technical Committee on Data Engineering**, 16(2), June 1993. IEEE Computer Society.

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