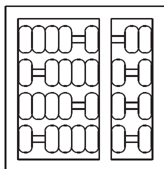

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Synchronization and Disconnected Operation in WorkToDo

M. Beatriz F. Toledo *Gustavo Kasprzak*
Leonardo H. Reinehr *Hudo R. Almeida*
Itana M. S. Gimenes

Technical Report - IC-03-006 - Relatório Técnico

April - 2003 - Abril

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Synchronization and Disconnected Operation in WorkToDo

M. Beatriz F. Toledo*, Gustavo Kasprzak, Leonardo H. Reinehr, Hudo R. Almeida
Itana M. S. Gimenes†

Abstract

This paper presents WorkToDo, a workflow management system compliant with the WfMC Reference Model. It addresses WorkToDo distributed architecture, its prototype implementation based on CORBA, mechanisms for task synchronization and support for operation in wireless environments. In this type of environment, users should be able to perform tasks independently from location and type of connection. To achieve these goals, the proposed workflow system supports task reservation and anticipated transfer of data and applications to the mobile computer. Other issues related with the concurrent execution of tasks are also considered. Synchronization mechanisms are included to avoid inconsistencies that may arise from the interleaved execution of tasks. Moreover, WorkToDo distributed architecture includes features to achieve a great level of reliability and scalability.

Keywords: Workflow Management System, Mobile Computing, Wireless Communication, Task Synchronization, CORBA.

1 Introduction

Workflow Management Systems have proved to be an important tool for many types of organizations. However they still have some limitations concerning issues such as interoperability, scalability, reliability, synchronization and flexibility of use [2, 9, 13, 14, 20].

In this paper, we address most of these issues in the context of WorkToDo, a WfMS compliant with the WfMC Reference Model [10]. WorkToDo distributed architecture was designed to provide a great level of reliability and scalability.

Another one of its goals was allowing users to execute tasks even when they are disconnected or in motion. For this, it provides features such as task reservation and anticipated transfer of data and applications to the mobile computer. Moreover, the concurrent execution of tasks may cause inconsistencies when they share resources. To avoid these inconsistencies, WorkToDo includes synchronization mechanisms to guarantee mutual exclusion of incompatible tasks and ordering requirements between workflow processes.

*Institute of Computing, University of Campinas, 13081-970 Campinas, SP.
†Department of Informatics, Universidade Estadual de Maringa, Av. Colombo 5790, 87020-900, Maringa - PR.

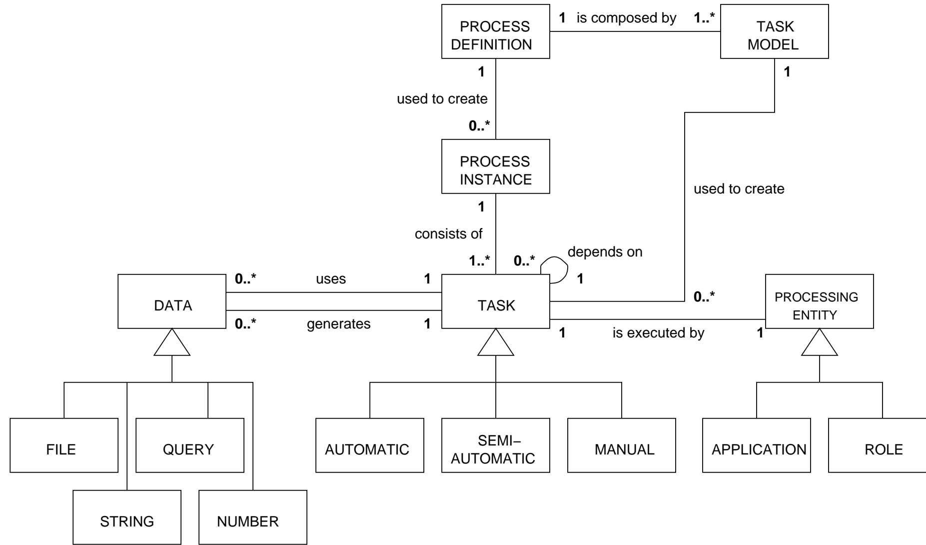

Figure 1: The WorkToDo process model.

The paper is organized as follows. Section 2 describes the WorkToDo process model. Sections 3 and 4 presents its basic functionalities and distributed architecture respectively. In Section 5, support for operation in wireless communication environments is discussed. Mechanisms for task synchronization are presented in Section 6. Some details about the prototype implementation are described in Section 7. Related work is presented in Section 8 and Section 9 ends the paper with conclusions.

2 Process Model

When modeling a process we define the following aspects: which tasks should be executed, in which sequence, by whom and with what data. These are known as functional, behavioral, organizational and informational aspects of a process, respectively [12]. These aspects are described below.

Figure 1 shows a representation of the model used in WorkToDo. A process instance is created from a process definition. It is a set of tasks and dependences among them; each task may use a set of input data and generates some output data with a processing entity responsible for its execution.

2.1 Functional Aspect

The set of tasks composing a process defines the process functional aspect. Tasks are defined by *Task Models* in which characteristics such as task class (for synchronization purposes), if a task can be executed in disconnected mode, priority, deadline and type are specified. Task types are the following:

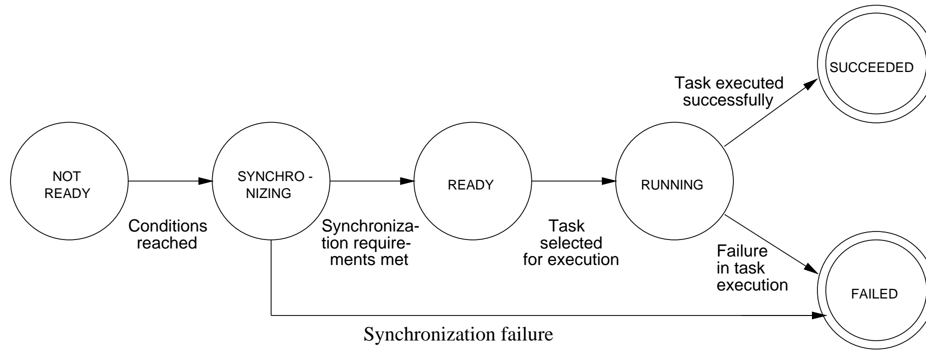

Figure 2: State transition diagram for a task.

Automatic. Tasks which are executed by a software invoked automatically by the WfMS (e.g., accessing a database).

Semi-automatic. Tasks which are executed by a human assisted by a software (e.g., writing a document using a text editor).

Manual. Tasks which are executed exclusively by a human (e. g. sending a letter) but whose success or failure must be notified to the WfMS.

WorkToDo defines five possible states for a given task: `NOT_READY` – the task cannot be executed because one or more conditions for its execution have not been reached yet; `SYNCHRONIZING` – all the conditions expressed by dependence rules (described in 2.2) are satisfied; `READY` – synchronization conditions are satisfied and the task is ready for execution; `RUNNING` – the task is being executed; `SUCCEEDED` – the task executed with success; `FAILED` – a error occurred during task execution. Transitions between states are shown in Figure 2. `NOT_READY` is the initial state, `SUCCEEDED` and `FAILED` are the final states.

2.2 Behavioral Aspect

A dependence rule contains the conditions that must be reached to allow the execution of a task. These conditions are described in terms of task state transitions. A rule is composed by zero or more terms containing a task name and a state. These terms may be operands of boolean operators (`and`, `or`).

An example of dependence rule is shown below:

$$\texttt{and } (T_1 \rightarrow \texttt{SUCCEEDED}, T_2 \rightarrow \texttt{SUCCEEDED})$$

In this example, the task associated with the rule will be executed only when tasks T_1 and T_2 reach the `SUCCEEDED` state. The dependence rule has two terms: $T_1 \rightarrow$ `SUCCEEDED` and $T_2 \rightarrow$ `SUCCEEDED`, which are operands of an `and` operation. Valid states for terms are those valid for tasks. Tasks with empty dependence rules enters the `SYNCHRONIZING` state.

2.3 Organizational Aspect

Every task has a processing entity responsible for its execution. Processing entities are specified in the task models. The WorkToDo defines two types of processing entities:

Applications. These are the processing entities for automatic tasks. When the task is initiated, the correspondent application must be invoked (possibly with parameters and/or input/output data). Each automatic task has an associated application.

Users. Users are the processing entities for semi-automatic and manual tasks. Tasks are assigned to users through a role mechanism. A *Role* is a group of users with some set of characteristics. The task model specifies a role R, allowing any user belonging to R to execute the task.

2.4 Informational Aspect

Tasks can use input data and generate output data. In WorkToDo, data may be: Files – files stored on disk; Queries – SQL (Simple Query Language) expressions which are queries on databases that return a set of tuples; Strings – alpha-numerical values; and Numbers – integer or real values.

2.5 Process Definition Language

WorkToDo provides a *Process Definition Language* (PDL) to allow the specification of process definitions, task models and application definitions. A process definition contains a list of tasks and, for each task, its input and output data, its dependences and its processing entity. A task model specifies the task characteristics, such as type and priority. An application definition contains the application characteristics, such as size and operating system on which it can be executed.

3 Basic Functionalities

There are two types of users defined in WorkToDo that interact with the system through a graphical interface. The user types are the following:

Administrator. An administrator has some privileges of administrative and management nature. The administrator is actually a role which can be assumed by many users. The operations available for the administrator are described in Table 1.

Participant. A participant may create new process instances and execute semi-automatic and manual tasks. Every process instance has a user in charge, who must take certain decisions when necessary. The WfMS notifies this user about task deadlines, task execution failures, start and end time of instance execution; the user in charge decides what to do in such situations (e.g., re-execute a failed task). The operations available for participants are described in Table 2.

Table 1: Description of operations available for administrator.

Operation	**Textual Description**
CreateProcess	Creates a process instance.
InterruptProcess	Interrupts a process instance.
ResumeProcess	Resumes a process instance.
CancelProcess	Cancels a process instance.
InsertTask	Inserts a task into a worklist.
RemoveTask	Removes a task from a worklist.
Query	Queries information about users, roles, process definitions, task models, application definitions, synchronization classes, ordering definitions and other configuration aspects.
Update	Updates information about users, roles, process definitions, task models, application definitions, synchronization classes, ordering definitions and other configuration aspects.

4 Architecture

WorkToDo has a distributed architecture, shown in Figure 3. Each component is described below.

Table 2: Description of operations for participants.

Operation	Textual Description
VisualizeWorkitems	The worklist shows the user's workitems, listed according to one of four ordering types: by arrival time, by priority, by deadline and by size of related data. A user can also move workitems to any position in the worklist.
SelectWorkitems	When the user wants to execute a workitem, he must select it. Upon selection, this workitem is removed from other users' worklists. The user who selected the workitem is then authorized to execute the workitem.
LockWorkitems	When the user wants to execute a workitem in disconnected mode, he must lock it (as will be seen in Section 5). Thus, the workitem is removed from the worklists of the other users of that role, in a similar way as if the workitem had been selected.
CreateProcess	A user is also allowed to create instances of a process when he belongs to the creator role of this process definition.
CheckProcess	Returns the list of executing processes
QueryState	Queries the state of processes in which he participates

4.1 User and Role Manager

The User and Role Manager (URM) is responsible for managing users and roles. It provides the following operations: insertion/removal of users to/from roles, retrieval of a user's roles, retrieval of users in a role, among others. The URM controls two repositories storing information about users and roles.

4.2 Definition Manager

The Definition Manager (DM) manages process definitions, task models and applications definitions. The DM provides operations for retrieval of existing definitions and insertion/removal of definitions. The DM manages three repositories which stores process definitions, task models and applications definitions.

When a process definition is inserted into the corresponding repository, the DM asks for a role name. This role is called the process *Creator Role*. Instances of this process definition can be created only by users belonging to the corresponding creator role.

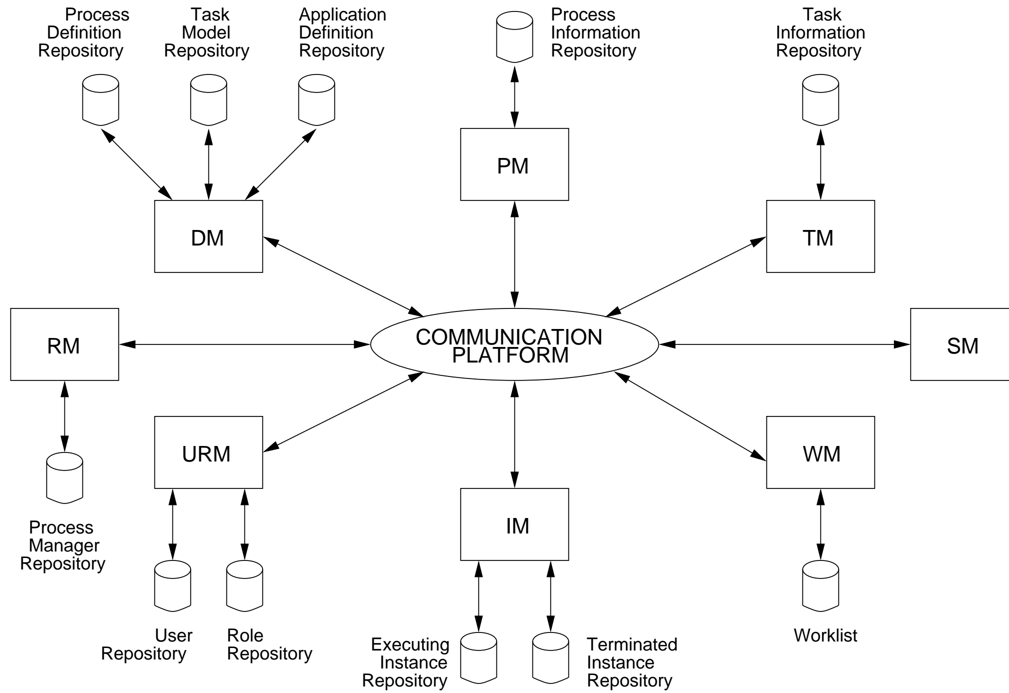

Figure 3: WorkToDo distributed architecture.

4.3 Instance Manager

The Instance Manager (IM) maintains information about the process instances in execution, as well as those already executed. The IM is also responsible for creating a Process Manager for each new instance; at creation time, the IM transfers a copy of the process definition to the Process Manager. The IM maintains two repositories with information about executing and terminated instances.

When a new Process Manager is created, it is registered with the Recovery Manager (described in 4.7).

4.4 Process Manager

The Process Manager (PM) is responsible for coordinating a process execution. The PM verifies which tasks are ready to execute, initiates their execution and collects their results, verifying if these results allow the execution of other tasks. There is one PM for each executing process instance.

The PM maintains a list with the tasks that compose the process, called *Tasklist*. The tasklist stores all the information needed for the execution of the tasks, such as name, type, input and output data, processing entity and current state. This information is called *Task Definition*. The tasklist consists, therefore, of a list of task definitions. The PM also maintains a repository which stores the current state of the instance, containing information such as the instance name, its start and end time and its tasklist.

The PM contains two sub-components:

1. **Scheduler.** This component evaluates the conditions for task execution. Every time a state transition of a task T occurs, the Scheduler reevaluates the dependence rules of the tasks related to T. When the conditions of a dependence rule are all satisfied, the task state is modified to `SYNCHRONIZING`.

2. **Dispatcher.** This component prepares the tasks for execution. The Dispatcher periodically checks the tasklist for ready tasks. When a ready task is found, the Dispatcher executes one of the following actions: if the task is automatic, it creates a Task Manager to manage the task execution; if the task is semi-automatic or manual, it notifies the users belonging to the task role about its availability. In the case of semi-automatic and manual tasks, the users to be notified are chosen according to a *Notification Policy* defined at instance creation time such as, for instance, all users belonging to the task role.

It is also responsibility of the PM to manage possible failures of the Task Managers. In these cases, the PM can create a new Task Manager, in the same host or another, initializing it with the last known state of the failed Task Manager.

4.5 Task Manager

The function of the Task Manager (TM) is to control the execution of an automatic task. The TM is created by a PM and initialized according to the respective task model. The TM then invokes the application responsible for the execution of the task (with the necessary data) and monitors its execution. When it is finished, the TM notifies the correspondent PM, returning the task termination type (with success or with failure). If a task execution fails, the TM can execute the task again until it succeeds or it reaches a maximum number of retries (specified in the task model).

4.6 Worklist Manager

The Worklist Manager (WM) controls and maintains a list with all the tasks that can be executed by a given user. This list is called *worklist* and the tasks are called *workitems*.

Each workitem corresponds to some process task. Thus, every time a workitem state changes, the state of the corresponding task is also changed. The workitems correspond to semi-automatic or manual tasks.

There is one and only one WM for each user, that is executed in the user's machine.

4.7 Recovery Manager

The Recovery Manager (RM) is the component responsible for recovering failed PMs. For this, it checks periodically if all the registered PMs are still alive. When it discovers that a PM has failed, the RM reinitializes it in another host with the last consistent state before the failure.

To perform the recovery, the RM uses a Process Manager Repository that stores the process state in a persistent way.

4.8 Synchronization Manager

The Synchronization Manager (SM) prevents the concurrent execution of incompatible tasks. It is described in Section 6. The SM verifies for tasks in the `SYNCHRONIZING` state if their synchronization requirements are met. When this is true, the task is put in the `READY` state.

4.9 Considerations

The WorkToDo system is conformant with the WfMC Reference Model [10]. Scheduling and controlling of tasks is done by the Process Manager which incorporates the functionality of the workflow engine. The interface with users and applications provided by Worklist Managers and Task Managers corresponds to Client Applications and Invoked Applications respectively.

With respect to its architecture, WorkToDo was designed to be distributed in order to achieve a greater level of scalability and reliability. Therefore we chose to have one Process Manager to control only one process instance as it is the component with the heavier work load in the system. Moreover, the Synchronization Manager is implemented as a set of managers – one for each task class. The distributed implementation of the SM requires more messages among the set of managers but achieves better scalability. Greater reliability is guaranteed by monitoring TMs and PMs and restarting them in case of failures.

5 Disconnected Operation

This section describes operation in disconnected mode [17] which allows a user to execute tasks without being connected to the WfMS. To achieve this, the user chooses one (or more) workitem(s), disconnects from the system and after executing the workitem (minutes, hours or days later) connects again to the WfMS to notify the results of the execution.

From the WfMS point of view, at a given time a user can be *connected* or *disconnected*. A connected user is able to receive notification about ready tasks, to select tasks for execution, to create process instances and query instance states, among other operations. A disconnected user is restricted to the interactions with the Worklist Manager, which runs in the user's machine.

WorkToDo requires the user to notify the system prior to disconnection. Users must reserve workitems to execute in disconnected mode through a *locking* mechanism: when the user wants to execute a workitem in disconnected mode, he must lock it. By doing this, he reserves the workitem to himself, preventing other users to execute it. A workitem can only be locked if its correspondent task was defined as allowed to be executed disconnected.

During disconnected operation a user can visualize and modify his worklist as usual, besides executing workitems in the same way that when connected. The only difference is that

in disconnected mode the worklist shows only those workitems locked before disconnection; the remaining workitems get disabled.

Mechanisms for disconnected operation are described in 5.1. Extensions to allow operation in wireless environments are discussed in 5.2 and 5.3.

5.1 Mechanisms

When we allow disconnected operation, some new issues arise, requiring special attention from the WfMS. These issues are described below.

5.1.1 Task Deadlines.

Tasks may have deadlines, which represent the time limit to their execution. The deadline of a task is specified in the task model.

When the deadline is approaching, WorkToDo sends notifications to the user who selected the task. Moreover, the user in charge also receives notifications when the time gets closer to a certain limit. This allows him to take appropriate actions such as cancelling the process or executing the task himself (in this case the results brought by the original user are discarded).

The deadline of a task is controlled by the PM to which the task belong. However, if a user locks a task and becomes disconnected, his WM will control the deadline instead of the PM.

5.1.2 Disconnection Protocol.

In order to execute a workitem in disconnected mode, data related to the workitem must be available locally. Thus, the workitem must be *copied* to the user's machine, where copying a workitem means copying both data and application related to the workitem (if any). When the copying is completed, the workitem is said to be *transferred*. The steps to enter the disconnected mode of operation are the following:

1. The user notifies his WM that he wants to disconnect from the WfMS, specifying the maximum time he can wait for disconnection;

2. The WM checks if the user has some selected workitem. In this case the WM asks the user whether disconnection should proceed or be cancelled;

3. The WM notifies the URM about the user's disconnection;

4. The URM sets the user's status to disconnected;

5. The WM calculates the size of each locked and non-transferred workitem, defined as the sum of the size of the application and the amount of data related to the workitem (if any);

6. The WM calculates the amount of data to transfer, defined as the sum of the sizes of all locked and non-transferred workitems;

7. The WM verifies if the network bandwidth supports the transfer of the required data within the time specified by the user. In negative case, the system indicates the required time and allows the user to unlock some workitem(s), returning to step 5;

8. The WM verifies if there is enough free disk space in the user's machine to store the required data. In negative case, the system asks the user to correct the problem (e.g., freeing disk space) or to unlock some workitem(s), returning to step 5;

9. The WM copies associated data (and application) for each locked and non-transferred workitem;

10. The WM disables all non-locked workitems and notifies the user that he can disconnect.

Disconnections may occur at any time during the protocol execution, both by user's decision or by failures. After reconnection the data transfer is continued from where it was stopped. Moreover, a locked and transferred workitem can be executed in disconnected mode even if the disconnection protocol has not been executed entirely.

5.1.3 Reconnection Protocol.

When the user reconnects, the WfMS must be notified about the results of any workitem that was executed and any data generated during execution must be copied back to the WfMS. This is accomplished by the reconnection protocol, described below:

1. The user notifies his WM that he wants to connect to the WfMS;

2. The WM notifies the URM about the user's connection;

3. The URM sets the user's status to connected;

4. The WfMS notifies the user that he is connected and that the results of the workitem executions will be transferred to the WfMS;

5. The WM enables all workitems;

6. For each workitem executed in disconnected mode, the WM:

   (a) In background, sends the data generated by the workitem execution to the correspondent Process Manager;

   (b) Removes the workitem from the worklist.

In the moment of disconnection, all non-locked workitems are disabled; step 5 enables them again. Some of these workitems may have already been selected, locked or even executed by other users. Therefore, the worklist is updated in the next interaction between WM and PM or when the user attempts to select or lock a workitem. As with the previous protocol, this protocol execution may be interrupted by failures and resumed upon reconnection.

5.1.4 Considerations.

It is important to say that, even if disconnected operation is supported, not all tasks should be executed in disconnected mode [6]. Some aspects have to be considered, such as the amount of data and applications and the delay in task and process execution (because of the inherent delay of disconnected operation). Thus, it is necessary a trade-off analysis before allowing a task to be executed disconnected.

5.2 Prefetching

The prefetching technique (applied in the context of WorkToDo) consists of copying to the user's machine a workitem that later may be executed in disconnected mode. Thus, when the user chooses to disconnect, all information required for execution of workitems will be already available locally, minimizing the disconnection time. The prefetching is always made in background, transparently to the user, using available bandwidth.

The prefetching occurs as follows: while the user is connected to the WfMS, the WM periodically checks if the worklist contains some workitem which can be executed in disconnected mode. If it does, the WM copies the workitem to the user's machine. All workitems that can be executed in disconnected mode are included in the prefetching, but locked workitems are copied before non-locked ones.

5.3 Operation in Wireless Environments

WorkToDo allows users to interact with the system in a wireless environment where connections are unstable and bandwidth is low.

In the periods when there is little communication between the user's machine and the WfMS, the bandwidth can be used for prefetching. Thus, when the user selects a workitem, its data may be already available locally. In this case, it may be better to execute the workitem locally, avoiding communication with another machine and thus masking possible failures.

When the user selects a workitem, the WM verifies if the workitem has already been transferred. In positive case, the user may choose between local or remote execution. Local execution avoids remote communication, but requires data to be copied back to the WfMS. If the workitem execution has a high number of data accesses, the local execution will probably be better; on the other hand, if the execution generates a large amount of data, the time to transfer the results to the WfMS will be too high. This trade-off must be considered by the user to maximize efficiency.

If a connection breaks, the WM disables all non-locked workitems as well as the locked but non-transferred ones. But the user can keep working on the workitems that had been locked and transferred, exactly as in planned disconnection. If the user has some workitem being executed when the failure occurs, there are three cases to be considered:

1. If the workitem was being executed over local data, its execution may proceed as if the workitem were being executed in disconnected mode. The execution results will be transferred to the WfMS upon user's reconnection;

2. If the workitem uses remote data, its execution is stopped; when the user reconnects execution can be resumed;

3. If the workitem uses a remote application, after reconnection the user can restart the application and resume the task execution.

6 Task Synchronization

This section describes the synchronization mechanisms within WorkToDo. The first mechanism described in 6.1 is based on a software component [7] called *Synchronization Manager* which provides *mutual exclusion* capabilities when executing concurrent conflicting tasks. The other mechanism described in 6.2 guarantees ordering requirements between tasks from distinct process instances.

6.1 Mutual Exclusion Requirements

The *Synchronization Manager* (SM) described here prevents concurrent execution of conflicting tasks that could cause data inconsistencies. Each task must be associated with a class that determines the synchronization requirements for that task. The clause `TASK_CLASS` in the WorkToDo task model is responsible for this definition. The compatibility information amongst task classes is provided as a list of conflicting classes. Tasks of conflicting classes may not be executed concurrently in order to guarantee a controlled access to resources.

Two kind of actors can interact with the SM: human and software actors. Thus, three scenarios of usage are possible:

1. Usage by WorkToDo administrator(s) for the creation and destruction of conflicting classes and SM configuration;

2. Usage by WorkToDo users for querying about SM configuration;

3. Interaction(s) with the process(es) manager(s) for querying about SM configuration and invocations of the synchronization operations.

The SM was developed considering a distributed environment, a large number of task classes and a large number of tasks within each class. It is composed of two major object classes: the classes *ConflictingClassManager* and *ConflictingClass*. The first class creates instances of the second and manages information about them including, for instance, the host where each conflicting class instance was created. Each of these instances executes at a fixed network node whose location must be known by each process manager requiring their services. A conflicting class manager instance is accessed through a specific interface — called *IConflictingClassMgt* — that contains the following operations: create, destroy, retrieve and retrieve all conflicting classes. This class is associated with the WorkToDo administrator (graphical) interface that uses it to control the SM.

The ConflictingClass class models the concept of task classes. In a given moment, an instance of the SM can have several instances of this class, each one representing a task class and running in distinct hosts.

According to the three described scenarios of usage, the operations of the ConflictingClass are distributed into three distinct interfaces: *IConflictingClassConfigAdm*, which contains the operations that the WorkToDo administrator can use to query and adjust the ConflictingClass configuration; *IConflictingClassConfigUsr*, which contains the operations that other WorkToDo users can use to query and adjust (in a restricted way) the ConflictingClass configuration; *ISynchronization*, which contains the operations related with task synchronization. The first two interfaces contain ordinary operations for querying and updating. Our focus in this section is on the presentation of the *ISynchronization* interface.

6.1.1 The ISynchronization Interface Operations.

The interaction between process managers that require task synchronization and an instance of the class ConflictingClass occurs through the operations *beginSync* and *endSync*. Table 3 describes these operations.

Table 3: Description of the ISynchronization operations.

Operation	Textual Description
beginSync	Just before scheduling the task for execution, the process manager responsible for the task must invoke this operation on the corresponding task class. The operation may return synchronization success or synchronization failure. It has one parameter specifying priority used to avoid starvation.
endSync	When the task finishes, the workflow engine responsible for the task must invoke this operation on the corresponding task class to confirm the task completion.

A process manager invoking the beginSync operation for a task A obtains *synchronization success* if there are no incompatible tasks under execution. If all conflicting classes confirm that there are not incompatible tasks under execution, A can be safely scheduled. Otherwise, the process manager receives a *synchronization failure* and A execution must be postponed. WorkToDo handles synchronization failures by re-invoking the beginSync operation until synchronization success is obtained or a maximum number of retries is reached.

When a task ends, the process manager responsible for it invokes the endSync operation on the suitable task class. If there are not other tasks of this class under execution, the class notifies all conflicting classes. Suspended requests invoked on these classes will now be able to proceed.

Deadlocks may occur as the operations of the ISynchronization interface are executed in a critical region. For example, consider Task Classes 1 and 2 that are incompatible and a concurrent request beginSync received by each. Task Class 1 tries to query Task Class 2

and vice-versa. Thus a deadlock situation occurs. The solution adopted here is based on *timeouts*. When a thread is created to answer a beginSync request, another thread called *monitoring thread*, is also created at the same moment. The monitoring thread is put to sleep and it remains in this state for a pre-defined period. When it wakes up, it verifies if the beginSync thread is still executing and, if this is true, it interrupts the beginSync thread execution and finishes.

Using this technique, when one of the monitoring threads (for Class 1 or 2) wakes, it detects that the beginSync thread is still executing and kills it. Thus, the other beginSync thread involved in the deadlock can continue its execution.

6.2 Ordering Requirements

Ordering requirements may be necessary when considering conflicting pairs of tasks from distinct process instances. This is analogous to the problem of guaranteeing serializability for a set of transactions [4] whose pairs of conflicting operations are executed in the same order.

In WorkToDo, ordering requirements may be defined for pairs of processes. Within this definition stored on an independent *relative order repository*, each pair of conflicting tasks must be specified. Its syntax is shown below:

```
WORKFLOW_DEFINITIONS <definition_1> : <definition_2>;
RELATIVE_ORDER{
<task_from_definition_1> : <task_from_definition_2>;
⋮
}
```

The clause `WORKFLOW_DEFINITIONS` is followed by the pair of process definition names. The clause `RELATIVE_ORDER` initiates a list of conflicting task pairs.

Each process instance has an associated data structure, called *Relative Order Information Table* (ROIT). Each ROIT row consists of a process definition name and a list of pairs of conflicting tasks. The ROIT high-level scheme is presented in Table 4. This table is initialized during the process instantiation and is used for updating the two lists associated with each task: a *notification list* and a *dependence list*. Considering a task T, the first list stores the tasks and their respective process instances which must be notified about task T completion. The second list stores the tasks and their respective process instances which must notify task T about their completion. A task may be executed safely only when its dependence list is empty.

For example, considerer the following ordering definition:

```
WORKFLOW_DEFINITIONS P_1: P_2;
RELATIVE_ORDER{
T_11: T_21;
T_12: T_22;
}
```

Table 4: ROIT high-level scheme.

Conflicting Definitions	Conflicting Tasks			Instance Tasks
Def_1	task_from_def_1	$\rightarrow$	$\rightarrow$	task_from_inst_def
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$\ldots$	$\rightarrow$	$\rightarrow$	$\ldots$
$\vdots$	$\vdots$			$\vdots$
Def_$< n >$	task_from_def_$< n >$	$\rightarrow$	$\rightarrow$	task_from_inst_def
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$\ldots$	$\rightarrow$	$\rightarrow$	$\ldots$

If task T_{11} (of P_1) is scheduled for execution, T_{11} is inserted into the dependence list of T_{21} (of P_2), T_{21} is inserted into the notification list of T_{11}, T_{12} (of P_1) is inserted into the dependence list of T_{22} (of P_2) and T_{22} is inserted into the notification list of T_{12}. When T_{11} is finished, a notification is sent to T_{21} that removes T_{11} from its dependence list.

7 Implementation Remarks

This Section describes some characteristics of the WorkToDo prototype built to test the system. Furthermore, the Section contains an example of a process.

7.1 CORBA & Java

In the implemented prototype, we used CORBA [16] (IONA's CORBA implementation OrbixWeb 3.1 [11]) and Java [19] (JDK 1.2).

The choice for Java was motivated by its property to be multiplatform, allowing the system execution on different hardware and operating systems. This is an important aspect in a workflow system, where many users access the system from various machines.

CORBA was chosen because it provides features such as interoperability, transparent access and persistence, making the development of distributed applications easier. Furthermore, it is a relatively stable technology and has been used successfully in many contexts.

The components of the architecture were all implemented as CORBA servers, communicating through a Local Area Network (LAN). The only exception is the Worklist Manager, which may use a wireless link.

7.2 Example of Use

To illustrate the use of WorkToDo, we will consider the example of a maintenance firm. The process for performing a requested service consists of the tasks below:

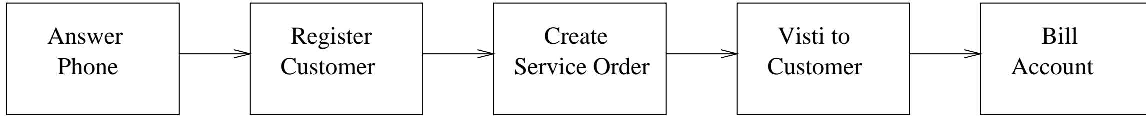

Figure 4: A maintenance process example.

Answer Phone. The office employee answers the customer's call. The request and customer's information are recorded. This is a manual task.

Register Customer. The customer's information is entered into the Customer Registry. This task is semi-automatic and cannot be executed disconnected. It belongs to the task class "UpdateCustomerRecord" which specifies mutual exclusion of tasks that access the resource CustomerRegistry.

Create Service Order. The office employee creates a service order for the new request. This is a semi-automatic task associated with a text editor application. Its output is a form filled with customer's name, customer's address and required service.

Visit to Customer. The technician visits the customer and performs the requested service. It can be executed disconnected. Its input data is the form generated by task Create Service Order and its output is the same form filled in with information about the service.

Bill Account. It is an automatic task executed by the billing system.

When paying the visit, the technician may detect that a part of an equipament is broken and may start another process to reserve the required item. For this, he must reconnect to the WfMS using his laptop and a cell phone. If he can finish the repair immediately he may remain disconnected and fill in the form with information about the finished service. When he later reconnects, the results of his work are transferred back to the WfMS.

8 Related Work

The Exotica Project [1] studies many issues related to WfMS such as the integration of advanced transaction models and WfMS, scalable architectures and disconnected operation [2]. Support for mobility is based on a locking mechanism. The user can select a group of tasks to be executed (through the *lock* operation, which removes the tasks from other users' worklists), causing the related data to be copied to the user's machine. While disconnected, the user can execute the selected tasks. Upon reconnection the data is copied back to the WfMS. Scalability is an issue also treated in [22].

In the INCAS model [3], the entities that execute tasks can communicate through wireless links. The INCAS only requires that entities are capable of receiving an INCA, executing the requested operations and redirect the INCA to the next entity. However, there are some implementation issues related to the last requirement for which no solutions have

been proposed yet (e.g., if an entity X must redirect an INCA to another entity Y, what X should do if Y can not be contacted for a long time?).

Bussler [6] presents a list of requirements to incorporate disconnected operation into conventional WfMS and proposes the *mobile worklists* concept to meet these requirements. There is not, however, any reference to some implementation. Jing *et al.* [13] discuss the prefetching technique for mobile environments.

The integration between CORBA and WfMSs is also explored by some projects [8, 22, 18]. These projects are mainly concerned with distribution, interoperability, scalability and dynamic reconfiguration.

In [5], the authors present the advantages of combining workflow management and transaction management and the use of a task compatibility specification to control concurrent executions of activities.

The work in [15] handles data inconsistencies arising due to concurrent executions and due to failures defining a uniform set of low-level mechanisms and dynamically modifying workflow rule sets for control flow.

The Contract model [21] is another work with interesting features related with reliability and synchronization. A Contract consists of steps which define actions and a script which defines the control flow between related steps. For reliability, concepts from transactions are applied. Invariants determine the conditions for step execution avoiding problems caused by step interleaving.

As Exotica, we have used the locking approach for the selection of tasks to be executed in disconnected mode. Moreover, we incorporate some of the requirements pointed out in [6], such as local management of task deadlines. But unlike other previous work, which have focused only on disconnected operation, our system also focuses on the issues of wireless environments such as partition failures and low network bandwidth. The synchronization of incompatible tasks is similar to the work in [5] but we use a coarser granularity (task classes). We also incorporate ordering requirements as discussed in [15].

9 Conclusions

In this paper we have presented WorkToDo, a WfMS compliant with the WfMC Reference Model. It focus on its distributed architecture, support for operation in wireless environments and task synchronization.

In WorkToDo, users can execute locked tasks from their portable device when disconnected or in motion. Prior to disconnection, WorkToDo executes a protocol which copies the task data to the user's machine. The tasks are then executed locally. Upon reconnection another protocol is executed, which transfers back to WorkToDo the task results. WorkToDo also implements a prefetching technique to optimize the copy of workitems. While the user is connected to the WfMS, the WM copies (in background) the workitems that can be executed in disconnected mode. If the user later locks these workitems, their data will already be available locally. Asynchronous techniques such as prefetching and postponed propagation of results can optimize the use of available bandwidth and mask eventual communication failures in wireless networks.

With respect to task synchronization, WorkToDo provides mutual exclusion capabilities when executing conflicting tasks. The other proposed synchronization mechanism guarantees ordering requirements between distinct process instances.

Moreover, issues such as reliability and scalability have been considered when designing WorkToDo architecture. Failed components are restarted and components such as PMs and SM were implemented to be scalable.

Finally, our approach aims not only to propose a model and architecture considering wireless communication environments, but also to validate the model presenting a CORBA-based prototype.

References

[1] Alonso, G., Agrawal, D., El Abbadi, A., Gunthor, R., Kamath, M., Mohan, C.: Exotica: A Project on Advanced Transaction Management and Workflow Management Systems. ACM SIGOIS Bulletin, vol. 16, n. 1 (1995)

[2] Alonso, G., Agrawal, D., El Abbadi, A., Gunthor, R., Kamath, M., Mohan, C.: Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System. In: Proceedings of the 3rd International Conference on Cooperative Information Systems. Vienna, Austria (1995) 99–110

[3] Barbara, D., Mehrotra, S., Rusinkiewicz, M.: INCAS: A Computation Model for Dynamic Workflows in Autonomous Distributed Environments. Technical Report, Department of Computer Science, University of Houston, USA (1994)

[4] Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley 1987

[5] Breitbart, Y., Deacon, A., Schek, H., Sheth, A., Weikum, G.: Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. SIGMOD Record ACM Special Interest Group on Management of Data, vol. 22, n. 3, (1993) 23–30

[6] Bussler, C.: User Mobility in Workflow Management Systems. In: Proceedings of the Telecommunication Information Networking Architecture Conference (TINA). Melbourne, Australia (1995)

[7] Cheesman, J., and John Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley (2000)

[8] Das, S., Kochut, K., Miller, J., Sheth, A., Worah, D.: ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for METEOR. Technical Report, Department of Computer Science, University of Georgia, USA (1997)

[9] Georgakopoulos, D., Hornick, M., Sheth, A.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. In: Distributed and Parallel Databases, Vol. 3 (1995) 119–153

[10] Hollinsworth, D.: The Workflow Reference Model, Workflow Management Coalition. In: D. Hollinsworth, The Workflow Reference Model, Workflow Management Coalition, TC00-1003, December (1994)

[11] IONA Technologies: OrbixWeb: IONA Technologies Java ORB Implementation. (2002). http://www.iona.com/products/orbixweb/index.html

[12] Jablonski, S.: Functional and Behavioral Aspects of Process Modelling in Workflow Systems. In: Chroust, G., Benczur, A. (eds.): CON 94 Workflow Management: Challenges, Paradigms and Products. R. Oldenburg (1994) 113–133

[13] Jing, J., Huff, K., Sinha, H., Hurtwitz, B., Robinson, B.: Workflow and Application Adaptations in Mobile Environments. In: Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications. New Orleans, Louisiana, USA (1999)

[14] Kamath, M., Ramamritham, K.: Bridging the Gap Between Transaction Management and Workflow Management. In: Proceedings of NSF Workshop on Workflow and Process Automation in Information Systems. Athens, Georgia (1996)

[15] Kamath, M., Ramamritham, K.: Failure Handling and Coordinated Execution of Concurrent Workflows. In: ICDE. (1998) 334–341

[16] OMG: The Common Object Request Broker: Architecture and Specification. Technical report, Object Management Group (1996)

[17] Reinehr, L. H., Toledo, M. B. F.: A CORBA-based Workflow Management System for Wireless Communication Environments. Proceedings of Confederated International Conferences: CoopIS, DOA and ODBASE (2002) 827–844. R. Meersman and Z. Tari

[18] Schill, A., Mittasch, C.: A Generic Workflow Environment based on CORBA Business Objects. Middleware'98

[19] Sun Microsystems. Java Development Kit, version 1.2 API Specification. (2002)

[20] Veijalainen, J., Lehtola, A., Pihlajamaa, O.: Research Issues in Workflow Systems. Technical Report, VTT Information Technology, Finland (1995)

[21] Wächter, H., Reuter, A.: The ConTract Model. Helmut Waechter, Andreas Reuter The ConTract Model chapter 7 in [4]. (1992) 219–263

[22] Wheater, S., Shrivastava, S., Ranno, F.: A CORBA Compliant Transactional Workflow System for Internet Applications. Middleware'98