International Journal of Cooperative Information Systems
Vol. 12, No. 4 (2003) 411–440
© World Scientific Publishing Company

# A FULLY DISTRIBUTED ARCHITECTURE FOR LARGE SCALE WORKFLOW ENACTMENT

ROBERTO SILVEIRA SILVA FILHO
*School of Information and Computer Science*
*University of California, Irvine, 92697-3430 Irvine, CA, USA*
*rsilvafi@ics.uci.edu*

JACQUES WAINER[*] and EDMUNDO R. M. MADEIRA[†]
*Institute of Computing, University of Campinas, 13083-970 Campinas, SP, Brazil*
[*] *wainer@ic.unicamp.br*
[†] *edmundo@ic.unicamp.br*

Standard client-server workflow management systems are usually designed as client-server systems. The central server is responsible for the coordination of the workflow execution and, in some cases, may manage the activities database. This centralized control architecture may represent a single point of failure, which compromises the availability of the system. We propose a fully distributed and configurable architecture for workflow management systems. It is based on the idea that the activities of a case (an instance of the process) migrate from host to host, executing the workflow tasks, following a process plan. This core architecture is improved with the addition of other distributed components so that other requirements for Workflow Management Systems, besides scalability, are also addressed. The components of the architecture were tested in different distributed and centralized configurations. The ability to configure the location of components and the use of dynamic allocation of tasks were effective for the implementation of load balancing policies.

*Keywords*: Large-scale workflow management systems; fully distributed workflow architectures; CORBA workflow implementation; and mobile agents.

## 1. Introduction

Workflow Management Systems (WFMSs) are used to coordinate the execution of a vast set of cooperative applications ranging from business processes, such as loan approval and insurance reimbursement, to large-scale software development projects and manufacturing control systems, to list some examples. Such processes are represented as workflows: computer interpretable descriptions of activities (or tasks), and their execution order. A workflow can also describe the data available and generated by each activity, parallel and optional execution paths, synchronization points and other aspects of the execution of complex inter-dependent cooperative tasks. Some of these aspects include policy constrains such as when the activities should be executed, a specification of who can or should perform each activity, and which tools and programs are needed during their execution.[8]

Many academic prototypes and commercial WFMSs are based on the standard client-server architecture defined by the WFMC (Workflow Management Coalition).[27] In such systems, the workflow engine, the core of a WFMS, is executed in a logically centralized server that typically stores both the application data (the data that is used and generated by each activity within the workflow), and the workflow data (its definition, the state and history information about each instance of the workflow, and any other data related to its execution). Even though, in most of such systems, different people or processes in their local hosts execute the workflow activities, a central server coordinates the control and data flow.

There are many reasons for a more distributed architecture. Firstly, workflow processes are inherently distributed: different processes may run in different parts of the organization (the travel reimbursement processes are different when performed in the North American branch and in the South Asian branch), or increasingly, a single case may run in different parts of the organization (part of a marketing campaign is executed at the headquarters of a company, whereas some of the activities of the campaign are executed in the South American branch, and the post production in the European Branch). In both cases, a single workflow server is inappropriate. Secondly, if the number of workflow cases increases, the central server may become a bottleneck, or even a single point of failure, that risks the whole business operation.[1] A distributed architecture is not the only solution for both problems: interoperability among different workflow systems can solve the issue of geographical distribution among processes and among activities within a single process; improvement in hardware and architecture (such as grid computing)[22] are practical solutions for the scalability problems. However, distributed workflow architectures are also potential solutions for both problems.

Different approaches, ranging from the completely centralized to radically distributed architectures have been studied in the literature.[7] In commercial workflows, however, more pragmatic solutions such as replicated servers and computer clusters are used to address the required levels of scalability and fault tolerance in organizations such as banks and multi-national corporations.

In a previous work, we introduced the WONDER (Workflow ON Distributed EnviRonment) architecture,[20] a radically distributed and configurable WFMS system, based on the mobile agent paradigm. A mobile software agent, or an agent for now on, is defined as an object that migrates through many nodes of a heterogeneous network of computers, under its own control, in order to perform tasks using resources of these nodes.[5,11]

In the WONDER architecture, the control, the storage of data, and the execution of activities are completely decentralized. Moreover, the component configuration is completely customizable and the workflow execution can be distributed over the many hosts of an enterprise computer network. WONDER provides a highly configurable and fine grained suite of components that can be arranged in different distribution scenarios, ranging from a complete centralized setting, where all the components are collocated in the same host, to a completely distributed setting, where each component is located at each host of a distributed system.

In this paper, we explore the use of different distributed workflow components configurations, analyzing its impact in the overall scalability of the workflow management system.

## 1.1. *Scenario*

To illustrate our approach, consider the following simplified example. ABC Financial Services has branches operating in many states, which are responsible for analyzing and approving thousands of finance applications on a daily basis. Loan applications represent a significant set of these proposals. Hundreds of requests are received every hour. These requests are issued electronically in the form of standard web forms, providing information such as the loan amount requested, the client information and the purpose of the loan. Each request (or case) is analyzed separately, in a different loan application process. Once received, the form is routed to an analyst in the credit department, who checks the client's credit history. This usually requires access to different credit databases. Once approved, the request is forwarded to another analyst, in the finance department, who calculates the appropriate interest rate and issues a personalized loan contract to the client. If a client has an insufficient credit history, the loan application can be rejected or, according to the analyst's criteria, an adjusted proposal can be issued, with a reduced credit amount. In both cases, a final proposal is then issued to the client, who can accept or reject it. If the proposal is accepted, a loan manager in the local branch of the company processes the request and issues a payment order in the name of the client. At the end, whether successful or unsuccessful, the whole process is archived for future reference. It is important to highlight that the activities can be performed in company branches scattered over different parts of the country. For example, while the credit department is in city A, the client branch may be in city B, and the services providing credit information may be hosted in a different country.

The workflow for this scenario is represented in Fig. 1 as follows. The Check credit history activity is subdivided in 3 activities in a sub-workflow that checks different financial institutions.

## 1.2. *Terms*

We will use, from now on, the following definitions. A **process definition** or a **plan** is described in terms of the WFMC primitives: sequencing, and-join, and-split, or-join, and or-split.[27] A **case** is an instance of a process. Thus, if loan approval is a process, then "Joe's Friday $10,000.00 loan request to buy a car" is a case. Processes are defined in terms of **activities** or **tasks** (boxes in Fig. 1), which represent a set of atomic actions performed by a single person or by a program. Activities can be executed sequentially or in parallel (in AND — mandatory — or OR — optional — branches). **Role** is the generic description of a set of abilities required to a person

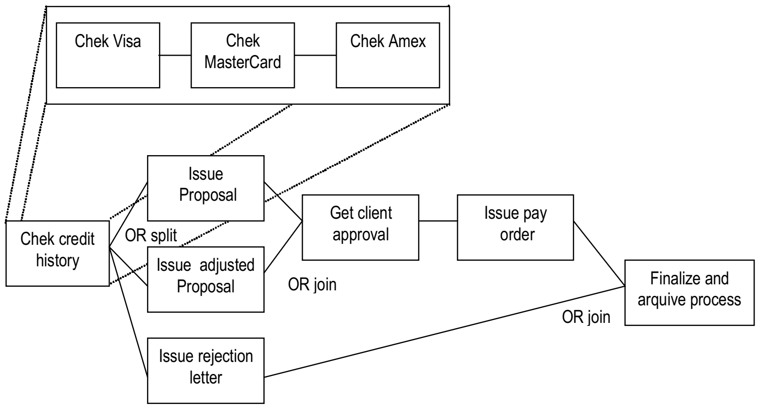

Fig. 1. Loan approval process description.

in order to perform an activity. Thus, credit analyst, finance analyst and branch manager are roles. People or programs that perform the activities are called **users** or **actors**, and a particular user can perform many roles. If the user is a person, she has a **preferential host**, a computer (or workstation) to where all her work related notifications and activities are sent. In particular, the notifications are sent to her **task list**.

## 1.3. *Large-scale WFMSs issues*

Because it is used to automate and coordinate intrinsically multifarious collaborative activities, the set of requirements that WFMSs must address is very broad. In particular, a comprehensive list of such requirements is discussed in Ref. 6.

In this paper, we focused on providing support for the following key requirements of large-scale WFMSs:

**Scalability:** The WFMS should not have its performance degraded due to the increase of: concurrent processes, cases or activity instances. It should also support a large volume of application data and/or large set of actors.

**Failure recovery:** The WFMS should be able to detect and deal with both software and hardware failures with the minimum user intervention as possible.

**Availability:** The system must not become unavailable or unreachable for long periods of time, especially due to failures or use load.

**Monitoring:** The WFMS should be able to provide information about the current state of all cases (and their activities) in execution.

**Traceability:** History (trace) information of currently executing or already terminated cases must be provided.

**Interoperability.** Different WFMS should be able to inter-operate through both inter- and intra-organizational boundaries.

**Support for external applications.** The execution of a particular activity may require external tools (such as word processors, spreadsheets, CAD systems, expert systems, and so on). The WFMS should be able to interface with these applications and determine when these tools have been terminated, managing the data read and produced by these applications.

**Load Balancing.** The human work as well as the processing power of the system as a whole should be allocated in a way that prevents excessive or idle use of such resources.

### 1.4. *Paper description*

The next section briefly presents the main components of the WONDER architecture. Section 3 discusses the implementation of this architecture using CORBA (Common Object Request Broker Architecture). In Sec. 4, the process language is discussed. Section 5 presents results obtained during the execution of performance tests with a system prototype, Sec. 6 describes related work and Sec. 7 presents some conclusions.

## 2. The Distributed Model

In general terms, the WONDER architecture is based on the idea of combining the workflow engine (or scheduler) with the task executor components. This approach removes the need of a central authority that orchestrates the workflow execution, which results in total decentralization of control, allowing the independent move of the task executor component. In this schema, the case (a workflow instance) is enacted by a set of mobile agents (or components) that migrate from host to host to perform the case activities. The agent encapsulates both, the case data (forms and documents) and the plan for that case (the process description). The mobile agent has its own micro workflow engine and moves itself to a particular user's host once it determines, typically at runtime, the user/host that will perform the next activity. Once the activity is finished, the agent identifies another user to perform the next activity and moves to his/her host carrying the data produced in that process step. The use of mobile agents provides autonomy and processing load distribution to the system, coping with the scalability requirement, since there is no central control or data server, and there is no performance bottleneck. It also allows the dynamic allocation of tasks to actors, coping with load balancing and, since it does not rely on any central control, and the execution is scattered over many hosts, it improves the fault tolerance of the whole system.

Some additional components were defined in order to support the operation of the mobile task executor agent and to deal with the other requirements. The plan is a generic description of the workflow process and does not specify a particular user as the performer of an activity. Instead, it defines activity executors in terms of roles. Consider the credit approval activity in our example; the plan will state that a "credit evaluator", and not a specific actor, should perform the activity of "credit approval". In order to cope with this requirement, a role coordinator component was defined. Each role coordinator instance contains information about a particular role. In the example above, the case queries the credit approver role coordinator, and asks about a user to perform that activity. Once the user is identified, the case moves to that user's preferential host.

Monitoring is also an issue in our architecture. How to determine, without broadcasting, the current state of the case, composed of many activities scattered over a computer network? This task is performed by many case coordinator component instances that keep track of each specific case status as it moves along. Each time the case moves to a new user's host, it sends a notification to its particular case coordinator, allowing this component to track the progress and current state of that case.

Another important problem for the mobile agent architecture is failure recovery. The distributed characteristic of the architecture introduces many failure-candidate points, but keeps the failure isolated from other processes. What happens to the case if the host where one of its activities is executing crashes? To deal with this failure scenario, some caching policies were specified. In the even of a crash while one activity's case is executing in the current host, a persistent copy of its last state is stored at the previous hosts visited by the activities of the case. As soon as the failure is detected, the case coordinator elects another host/user to restart or resume the process in a consistent state before the crash. Furthermore, in not very reliable networks, to improve fault tolerance, the case coordinator may direct hosts to transfer this information to a backup manager.

In its essence, the WONDER architecture is structured as a distributed hierarchy of monitoring and policy enforcement components that provides the support framework for the migration and execution of the mobile workflow executors, implemented as mobile agents. These auxiliary components correspond to the process coordinators (there is one per process definition), the case coordinators (one per case), the role coordinators (one per role), the backup manager(s) and others described in the next section.

Even though our approach of decentralized components eliminates the bottleneck of traditional workflow systems, the use of distribution may increase communication among the distributed components. We feel that in the case of the WONDER architecture, there is no significant increase in the amount of communication, as we discuss below. The case coordinator manages one specific instance of a process and receives very short asynchronous notifications from the mobile agents (activities). These notifications comprise only the agents' current status and destination host. On the other hand, the backup manager may receive large amounts of data, but this transfer is done asynchronously when network and server load permit. The only standard servers, in a client-server sense, are the role coordinators, which receive a query and must return an answer before the agent migration proceeds. However, the respective amount of information exchanged also is small, involving the sending of a short query and the return of a user identity as an answer. Therefore, since message exchanging is small and asynchronous, the communication overhead is not a problem.

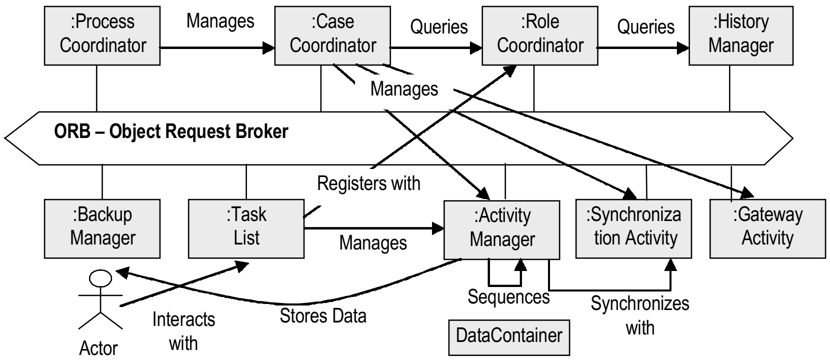

Fig. 2. The main components of the architecture.

## 2.1. *Main Components of the architecture*

The architecture, represented in Fig. 2, is composed of autonomous distributed components, which are described in the next subsections.

### 2.1.1. *Process coordinators*

Each process coordinator instance is responsible for the creation and management of case coordinators that monitors instances of a particular process definition. Upon a request for a new loan approval workflow, for instance, "the $2,000.00 loan application for the purchase of a computer", the "loan approval" process coordinator will create a new case coordinator for this particular request. This case coordinator will be responsible for the management of these particular loan application activities during the life time of such process instance.

Each process coordinator keeps track of all of its corresponding case coordinator instances in execution, being responsible for location, initialization, changing and termination of these cases. For example, if the general definition of a process is changed, say to introduce a new pre-approval credit activity, the corresponding process coordinator will update its own definition and will adopt these changes to all new cases to come.

### 2.1.2. *Case coordinators*

Each case coordinator tracks and manages the execution of a particular set of activities representing a process instance. Each case coordinator is responsible for detecting activities failures and for coordinating their recovery procedures. They execute the finalization procedures of their case, collecting cache information left by activity managers in the hosts of the system. They also collect and store the case data in the history managers. Each case coordinator also answers queries about the current state of a case, notifies the process coordinator when a case is terminated, as well as other management procedures.

Each case coordinator creates a synchronization activity for each and-join specified in the process definition, adding their addresses (or names) to the plan used to configure the first activity of the case (see Sec. 2.1.4).

A new case coordinator is created for each new process instance. Its location in the network can be specified at creation time, by the process coordinator, to comply with load balance policies. When the case finalizes, the case coordinator is terminated.

### 2.1.3. *Role coordinators*

Each role coordinator is a specific resource locator component. It is responsible for the identification of users qualified to perform a particular activity. It also periodically collects information about a particular set of users, such as the activities that they are currently executing and their current work load. With this information, a "finance analyst role coordinator" can answer queries like "Who is the least loaded analyst?" or "Who are the available analysts?" There is one role coordinator per role in the system.

Each role coordinator may also have access to the History Managers (which store information about completed cases), and to corporate databases. With the help of these servers, the role coordinator can answer queries like: "Who is the analyst with the most experience in that kind of loan?" or "Who was the analyst that approved that line of credit?"

In summary, the role coordinators are components responsible for selecting the user (or users) that will perform a particular activity. There is not much literature on user selection policies, but we can anticipate some useful policies, all of which can be computed with the proposed architecture: "choose randomly among the users that can fill the role", "choose the least loaded user", "choose in a round-robin way", and so on. We can conceive other policies that use historical information about all cases, such as choosing the user with most experience with that customer, for example.

### 2.1.4. *Synchronization activities*

And-joins and Or-Joins are a particular problem in our workflow model based on mobile agents. Each join of a case must be created before the case begins, otherwise a mobile agent would not know where to go when it needs to synchronize with other mobile agents executing in parallel branches of the same case. The synchronization activities of a case will wait for all notifications (and-join) or the first notification (or-join) from its preceding activities before starting the following (output) activities. For example, in the workflow of Fig. 1, synchronization activities have to be created representing the or-joins before the activities "get client approval" and "finalize and archive process". During these or-joins, once one of the two possible mobile agents of this example has finalized and moved its data and state to the synchronization activity, this component merges all case data, and composes a new single agent that will execute the next activity. In an and-join synchronization activity, all input agents have to arrive before triggering the sequencing of the next activity.

A synchronization activity may also wait for other synchronization signals, such as external events. Although that is not contemplated in the WFMC definition, one can conceive that, for example, a meeting can only take place after all its preparatory activities are completed (the input activities for the and-join), but it may also have to wait for an external event that informs that the meeting room is available. In this case, the synchronization activity would also wait for this external event notification before proceeding to the next activity.

The synchronization activities are created specifically for each case, being deactivated when the case is finalized. Errors related to the failure of synchronization activities in the middle of a case, as well as delays or the lack of synchronization of parallel executing branches, are handled by the case coordinator.

#### 2.1.5. *Task lists*

The user interface is implemented as a task list, similar to an e-mail client. Each task list notifies the user of new activities that she is supposed to perform. This allows the user to accept or to reject the incoming activity according to the current specified policy. Furthermore, the user task list is her main interface to the WFMS itself. It allows the user to customize its preferred external applications, the policies for sorting the coming activities, her preferential host, and so on. It also monitors the user activity and collects her current workload state, forwarding this information to the role coordinators. The task list also provides access to the user workspace, the set of files and data necessary to execute each activity. There is one task list component per user in the system.

#### 2.1.6. *History manager*

The history manager (or managers) is (are) front-end(s) for the repository of completed cases. When a case coordinator finishes its work, all relevant data used by the case are stored in the history repository. Such procedure allows for the cases to be audited and the memory of the cases to be archived for further review.

#### 2.1.7. *Backup manager*

The backup manager (or managers) is (are) front-end(s) for the repository of the intermediary state of the active cases. As we mentioned before, a copy (cache or backup) of the mobile agent execution state and the workflow data are stored in some of the hosts where the activity manager executed. These hosts are neither trusted to store this information indefinitely, nor to be active when this cached data is needed (in recovery procedures, for example). The backup manager runs in a more reliable and powerful machine. It collects the cached data left by the mobile agents, under the command of the case coordinator. Once the backup is performed, the state information can be erased from the users' hosts.

There may be many backup managers in the systems, one per process, one for a group of processes, or many for a single process. The identity of the backup manager and the moment to perform the backup is parameterized in the case coordinator. This decision is based on many parameters such as network and system loads. Once the backup is made, the user host can erase the past state information of that case.

### 2.1.8. *Activity managers*

Each activity manager is a mobile agent that executes and conveys the case data throughout the network of user's hosts. The mobile agent is implemented using a weak migration strategy[15]: there is no mobility of binary code between hosts, only the agent execution state and the necessary case data are transferred between hosts.

Each time it migrates, the activity manager coordinates the execution of an instance of an activity for a particular case. When the activity manager detects the end of the current activity, it initiates the migration process. Using the weak migration process, the current activity manager creates an instance of itself in the preferential host of the user that will perform the activity. This new activity manager instance is, then, configured with the next activity specific data, and the current activity case state.

Only the necessary data is transferred from one activity to another. The plan has the last location of all the case data in the form of links. This allows the current activity to fetch the necessary data for its execution. Once modified, a copy of the piece of data is stored in the local host of the current activity and its new location is updated in the plan.

Once created, the next activity is started at the point that the previous one had stopped. The previous activity manager is terminated and has its state and data saved to a special repository in the local host. The plan interpretation is resumed in the new host and the activity is performed using the appropriate applications, through the use of application wrappers. The recent created activity becomes the current one. This activity manager waits until the user finishes the activity execution and then computes who should execute the next activity (by interpreting the plan that came along with the case state and by querying the appropriate role coordinator). If the next activity is to be performed by a user, the activity manager sends the appropriate information to that user's task list, notifying the case coordinator that the activity has ended and who is the user selected to perform the next activity. After that, it transfers the case information to the recently created activity manager and the workflow execution is resumed. This process is repeated until the end of the case.

At the beginning of a case, the case coordinator creates the first activity manager for that process instance, which starts executing. This agent can clone itself in order to follow AND branches in the workflow.

#### 2.1.9. *Application wrappers*

An application wrapper is a component that controls the execution of a particular invoked application (office, CAD, database tools and so on). It launches the application with its necessary initialization parameters, together with the activity data, collecting the application output. It is a bridge between specific programs and the activity manager. When the task finishes, the Wrappers notify the corresponding Activity Manager that collects the generated data in the user workspace.

#### 2.1.10. *Gateway activities*

In order to address the WFMS Interoperability requirement, the gateway activity component was defined. It is responsible for bi-directional conversion of workflow data and control between two different WFMSs, defining a special bridge to external applications in the WONDER model. This component can, for example, intermediate the communication between two or more WFMSs using the Wf-XML data interchange format standard.[26]

## 3. CORBA Implementation

CORBA[16] was chosen as the middleware to support the WONDER architecture implementation, based on its ability to integrate heterogeneous applications and its high-level support for distributed object management whose use in the implementation of WFMSs is discussed in Ref. 7. CORBA also provides a set of services and communication transparencies that improve the distributed applications development. It specifies an object-oriented distributed bus, providing transparencies of access (independence of hardware, language or operating system) and location (independence of the host where the object is executing). It offers object-oriented programming advantages, such as inheritance, information hiding, reusability and polymorphism. It also enables the use of legacy applications, which were developed for different hardware and software platforms. This is possible through the definition of the IIOP (Internet Inter Orb Protocol) and the CORBA IDL (Interface Definition Language) that allows the generation of interfaces to a large set of programming languages.

Each component of the architecture, described in the previous section, was mapped to a particular CORBA object. In order to fully support our approach, some additional services had to be implemented on top of the standard CORBA implementation. A more detailed description of this mapping is described in Ref. 20.

### 3.1. ***References to CORBA objects***

The CORBA 2.0 standard IORs (Interoperable Object References) are not fully adequate for our application. IORs uniquely identify an object in the CORBA name space. These references are dynamically allocated by the ORB and include information such as the IP address and port number that respectively locate the access point to an object interface in a particular host.

Since the total execution time of a case may least many days, or even months (in a large software development process, for example), one cannot assume that, for a whole case execution lifecycle, an object (such as the synchronization activities or case coordinator) will be active, on the same port it was created, having the same IOR. Objects need to be deactivated when inactive for a long time, in order to allow the execution of other processes, or even due to host and connection failures.

Recently, the OMG (Object Management Group) finished the specification of the object persistency service. However, by the time of the implementation of the WONDER system, such service was not available. Thus, our own persistency service had to be developed. In our scheme objects are locally stored and identified using the following naming scheme: [host, process, case, actor, activity, file] for files; [host, process, case, actor, activity] for activities; [host, process, case] for case coordinators; [host, process] for process coordinators; [host, backup-server] for backup managers, and so on.

In order to provide transparent object persistence, each host has a special component called Local Object Activator (LOA). It executes as a hook in the WONDER runtime environment daemon (orbixd — OrbixWeb locator daemon) and intermediates the object creation (bind), activation, deactivation and persistence, saving the object state and data in a local reserved disk area (the object repository). For example, the case coordinator for a $500.00 loan approval (case C4375), of the process "loan approval" (process P12), in the host abc.def.com is identified by (abc.def.com, P12, C4375). To access this object (or formally to bind to this object), a process must send the reference (P12, C4375) to the LOA in the machine abc.def.com, which will activate and restore the state of that case coordinator. This activation uses the information previously stored in the object repository. The LOA then returns the new IOR of the restored object to be immediately used.

## 3.2. *CORBA services*

Many CORBA based Workflow architectures use a subset of the OMA CORBA Services.[17,24] The most commonly used services are the Naming, Event, Notification, Security and Transaction. Due to the large-scale requirements of the WONDER architecture, and its mobile object approach, some inadequacy points of these services came up. These issues are discussed as follows.

Some workflow implementations use the CORBA Transaction Service to coordinate the data flow among many different servers.[17,23] This approach creates a fail-safe data transfer protocol among different activities, providing a set of "transactional communication channels".

Large WFMSs usually require transactional semantics, but may not always require distributed transactions.[19] In the WONDER architecture the Activity Manager peers enforce a transactional semantic when the data is transferred. All the data and the case state are transferred simultaneously, in a single operation invocation, from one activity manager to another. During splits, this process is iterated for each activity in the branch. Hence, the CORBA method invocation mechanism is sufficient for our implementation. Errors at this time are handled using retransmission policies. If some error occurs during the remote operation invocation, due to a temporary link crash, for example, the ORB throws a *SystemException.* This exception is caught and resolved by the data sender which, according to the failure reason, can result in another method invocation when the link is up again. If failure persists, the case coordinator carries on the error handling procedure, creating an alternative path to be followed. This simple approach dispenses a more complex control implemented by a transaction server.

The CORBA Event and the Notification Services decouple the producer and consumer objects, implementing a message queue. These messages can be made persistent in some *COSNotification* proprietary implementations.[18] This safe event channel, however, increases the failure detection complexity: how can an activity manager identify the failure of a case coordinator if their communication is decoupled by the notification service?

The WONDER architecture does not rely on any standard CORBA naming service because of the IOR problems described in Sec. 3.1. Instead, each host executes an activation agent that resolves markers (OrbixWeb human-readable object names) to IOR object references, working as a local name service. This activation agent, operating with the LOA, is also used to implement the objects activation and deactivation, besides their persistence. The activation agent is implemented using the *OrbixWeb orbixd* daemon and an *OrbixWeb LoaderClass* hook, where specialization implements the LOA object.

## 3.3. *Execution scenarios*

In this section, some execution examples are presented. They emphasize the behavior of the main objects of the architecture, showing their communication and interaction. For simplicity, we will not represent the interaction with the LOA object in our diagrams. This interaction occurs each time an object is created, restarted or reconnected. The scenarios are described using the UML sequence diagram notation.

### 3.3.1. *Activity sequencing*

Figure 3 presents a typical example of an activity sequencing (or agent migration) procedure. Generic activity and case coordinator names are used. When the activity execution ends (sending messages 5 and 6), the activity manager AM2 starts the new activity sequencing process. The case coordinator CC1, executing in a different host, receives an "end of the activity" notification (6). The AM2 activity interprets the process plan and determines which activity will be performed next. The AM2 queries RC1 (the role coordinator for the role to execute the next activity — message 8), which selects an appropriate user for that task. The AM2 notifies the user about

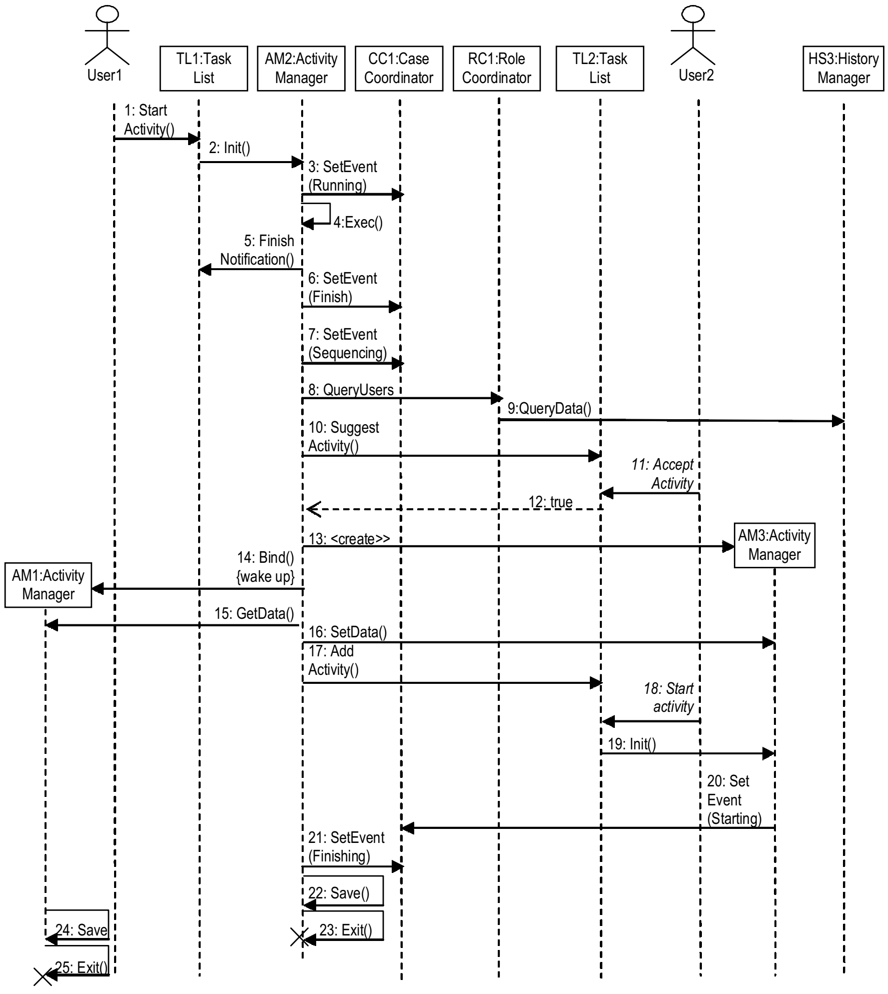

Fig. 3. Activity sequencing diagram.

the new activity. This message is sent to TL2 (10). If the selected user accepts the activity, the migration procedure starts (10 to 13). The activity manager AM2 requests the creation of the next activity manager, AM3, in the user's preferential host (13), and transfers all necessary data to this object (16). Since AM2 does not have all the necessary pieces of data to send to AM3 locally, it collects the necessary data files from AM1 (14 and 15). The data is wrapped in a data container together with the case state. Finally, the AM3 activity manager is inserted in the User2 task list (17). It is initialized (19) and the AM2 activity is finalized (21 to 23).

For performance reasons, only data necessary for the created activity is transferred. The remainder data are passed as links, in order to be retrieved by subsequent activities.

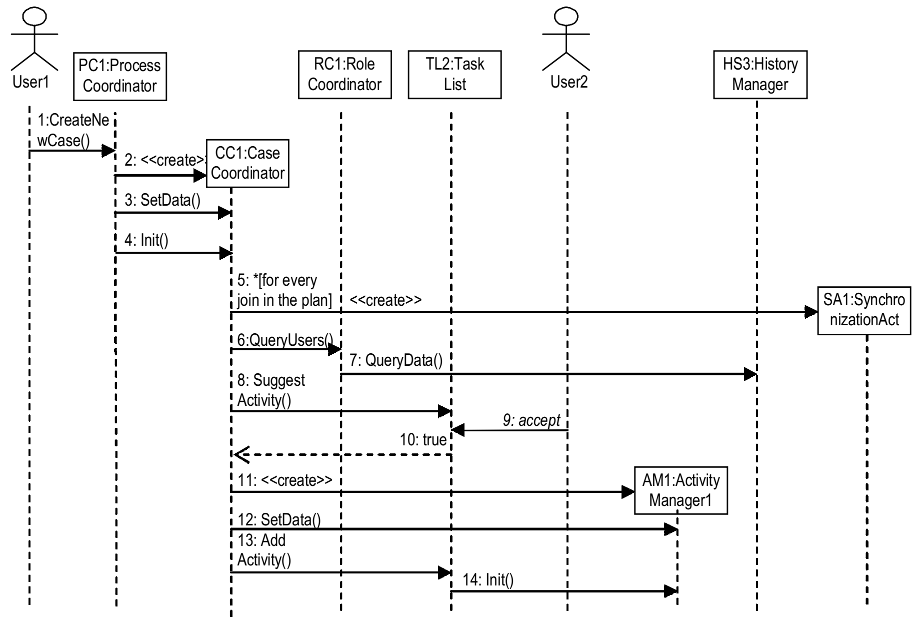

Fig. 4. Case creation sequence diagram.

## 3.3.2. *Case creation*

The case creation procedure, presented in Fig. 4, is initiated by a user (User 1) request in the process coordinator PC1 interface (1). This request results in the case coordinator CC1 creation and configuration (2 and 3). The setup process starts and the CC1 creates the synchronization activities for the case (5). After querying the RC1 role coordinator for a user to perform this activity (User 2), and after the activity acceptance by this user (8 to 10), the CC1 creates the first case activity AM1 (11 to 14) and the case starts.

## 3.3.3. *Activities and-split*

The and-split is implemented as a parallel sequence of activities; the procedure described in Fig. 3 is iterated for each activity in the branch. The newly created activities follow independent paths until a synchronization activity (and-join) is found.

## 3.3.4. *Activities synchronization*

The synchronization activities are created by the case coordinator, and their localization is placed in the process plan at the beginning of the case. When an activity ends, and its following activity is an and-join, the plan will have a reference to this synchronization activity address.

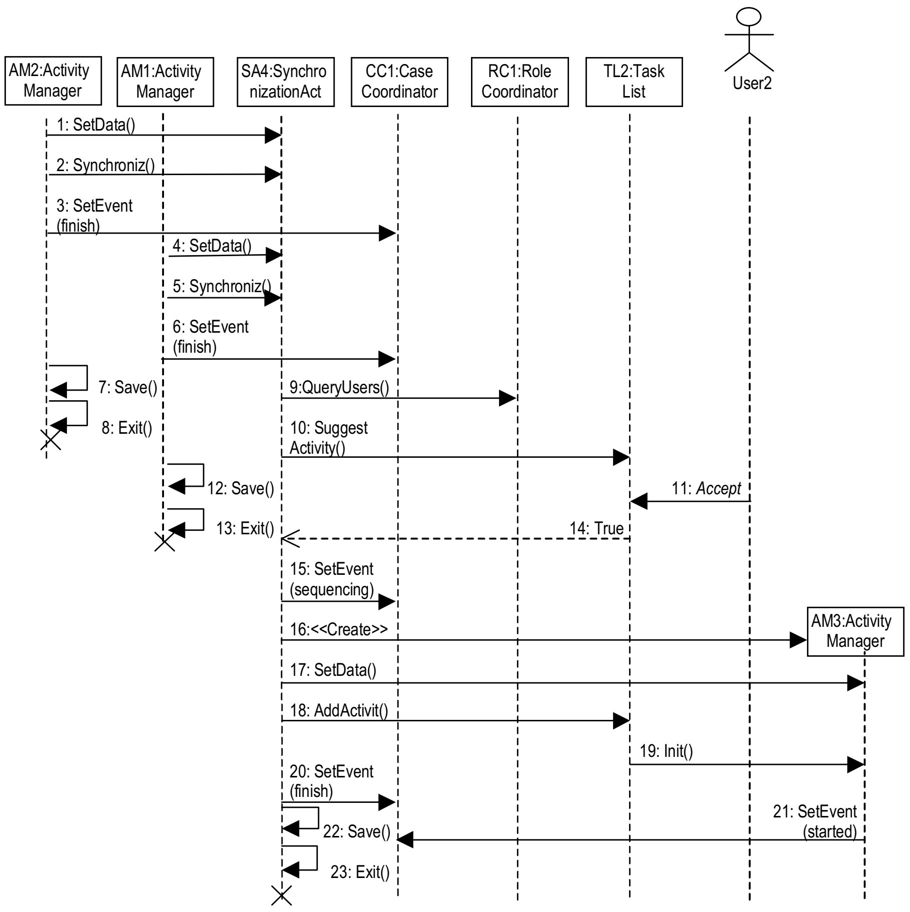

Fig. 5. An and-join synchronization diagram.

The synchronization procedure involving the activities AM1, AM2, and SA4 is described in Fig. 5. During this synchronization process, each activity manager notifies the synchronization activity SA4 and the case coordinator CC1 (2 and 3). After both activity managers (AM2 and AM1) have notified SA4, it starts the following activity in the standard way as described in Sec. 3.3.1. As usual, CC1 is kept informed of the progress of the case, manages the case and handles its failures.

### 3.3.5. *Case finalization*

Figure 6 presents the sequence diagram of a case finalization procedure. The case coordinator CC3, by the end of each case, removes the data stored at each host that executed at least one activity of the case, as well as all case data stored in the backup manager(s) (9, 11 and 13). An execution summary containing relevant data for future queries is stored in the History Manager HM2 (12).

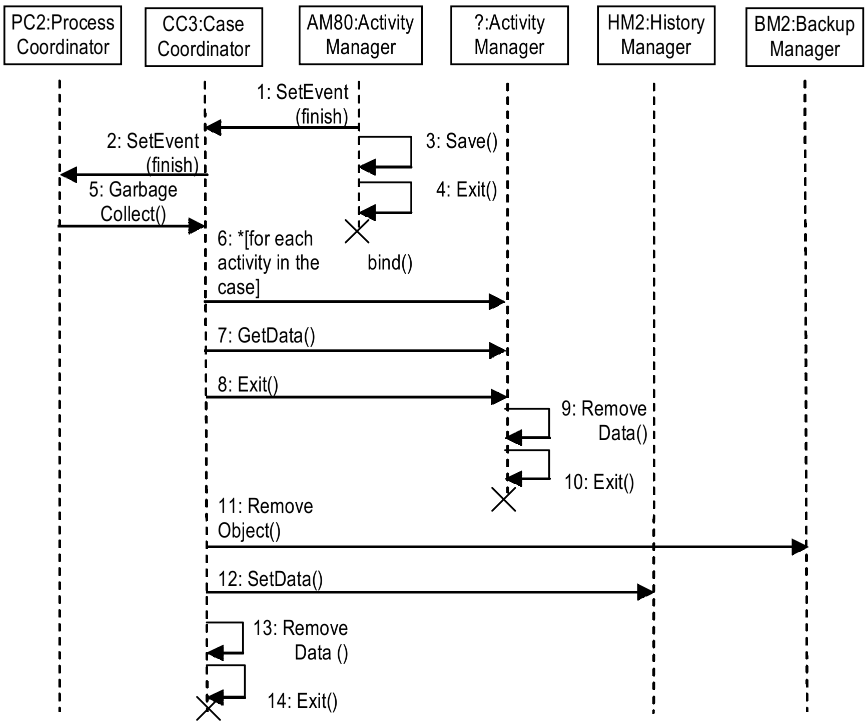

Fig. 6. A sequence diagram of a finalizing case.

### 3.3.6. *Failure recovery*

The case coordinator controls the failure recovery process. Whenever an activity stops sending notifications, or a host failure is detected during periodic polling of the most recent activities in the case, the case coordinator starts the recovery process. It consists in: halting the current process (current executing activities), restoring the system to a previous stable state, modifying the case process definition (adding compensation activities), and finally resuming the case. The Case Coordinator manages this routine using data stored in the object repository of each host, and in the backup managers scattered over the system.

Failures in the synchronization activities of a process instance are handled in a similar way. A new synchronization activity is created in another available host of the system and the agents are notified of the change. Their process definition is then updated to reflect the change. One problem that may occur is the failure of the synchronization activity in a partially synchronized state. In a branch with two activities, for example, the first may have already synchronized when the activity fails. In this case, the case coordinator will use the cached information in the previous host (of the synchronized activity) to reinitiate the synchronization process of this activity, after the creation of the new synchronization activity.

## 4. Process Description Language

As described in previous sections, the workflow management engine is a component of the activity manager that interprets the plan for the current case. The plan is described using a LISP-like language called PLISP (Process LISP). Since the central research question of the paper was to determine the behavior of the architecture components and the overall system in different distributed configurations, a simplified process language was defined.

PLISP supports the basic workflow control constructs as sequences and splits (joins are implicit). It also defines activities and its components: wrappers and data elements. The activity definition allows the optional specification of the location

```
(workflow st1

  (options
    (garbageCollect false)
  )

  (declarations
    (data data1
       "/home/msc98/931680/Data/teste1.doc" reference)
    (data data2
       "/home/msc98/931680/Data/teste2.doc" file)
    (data data3
       "/home/msc98/931680/Data/teste3.doc" reference)
    (data data4
       "/home/msc98/931680/Data/teste4.doc" file)
    (data data5
       "/home/msc98/931680/Data/teste5.doc" file)

    (application ap1
       "/n/dtp/StarOffice5.1/bin/soffice")
    (wrapper wp1
       ap1 (read data1 data2 ) (create data4) (modify data3) )
    (wrapper wp2
       ap1 (read data3 data2 ) (create data5) (modify data1) )
    (wrapper wp3
       ap1 (read data1 data2 data3 ) null (modify data5) )

    (activity act01
       "iguacu" "role1" "role query" "priority" "deadline" "descript1" wp1)
    (activity act02
       "iguacu" "role2" "role query" "priority" "deadline" "descript2"
       wp1 wp2)
    (activity act03
       "iguacu" "role3" "role query" "priority" "deadline" "descript3"
        wp1 wp2 wp3)

  ) // declarations

  (sequence sq1
     act01
    (andsplit split1
       (sequence sq2 act01 act02)
       (sequence sq3 act01 act03)
     )
     act02
   )
)
```

Fig. 7. A PLISP process template example.

(hostname) where it will be created, the role or the user to perform it, a custom query to be performed in the role coordinator, a priority value, a deadline, a short description and a set of wrappers to execute. Each wrapper can define a set of data elements created, read and modified by it. An example template is presented in Fig. 7.

An auxiliary bootstrap application was also implemented in order to initially load the case, role and process coordinators in the hosts specified in the boot-up script.

## 5. Tests

This section describes the tests performed with a prototype of WONDER. The tests were designed to access the general behavior of the system, analyzing the scalability, load balance and delay characteristic of the architecture when operating in different distributed and centralized configurations.

For these tests, the core components of the architecture, including the Activity Manager, LOA, Synchronization Activity, as well as the Case, Process and Role coordinators were implemented. The remaining components were partially implemented in order to mimic the behavior of the real system.

The system was developed in Java (Sun JDK 1.1), using the Iona OrbixWeb 3.1c framework, a CORBA 2.0 compatible ORB implementation.

The test was performed using SUN OS workstations. In special, pairs of similar machines were used: two Sun Ultra 2 (252 MB RAM), two Sun Ultra Enterprise (512 MB RAM), and two Sun SPARCStation 4 (64MB RAM). These hosts were connected by a 10Mb Ethernet Local Area Network.

### 5.1. *Overhead tests*

In order to determine the influence of the WONDER runtime environment alone in the overall performance of the machines, we defined an experiment in which a case was executed in different distributed configurations. For this test, there were no external applications invoked by the case, nor any additional case data being used by the activities. For this set of tests, the time intervals described in Fig. 8 were collected. A comparative chart with the results is presented in Fig. 9. The test consisted in the execution of a single and simple case, with 20 consecutive activities in two different configurations, centralized and distributed. The test was repeated for every pair of identical machines used throughout the tests. In the centralized tests the activities of the case and the coordinators (process, case, role) all execute in a single separated host. In the distributed tests, the activities were programmed to alternate between a pair of equivalent hosts, so that consecutive activities execute in different machines. The coordinators were configured to execute in a third host.

The initial motivation of this set of tests was to determine in a detailed way, the delays associated to every phase of an activity life cycle in order to identify and

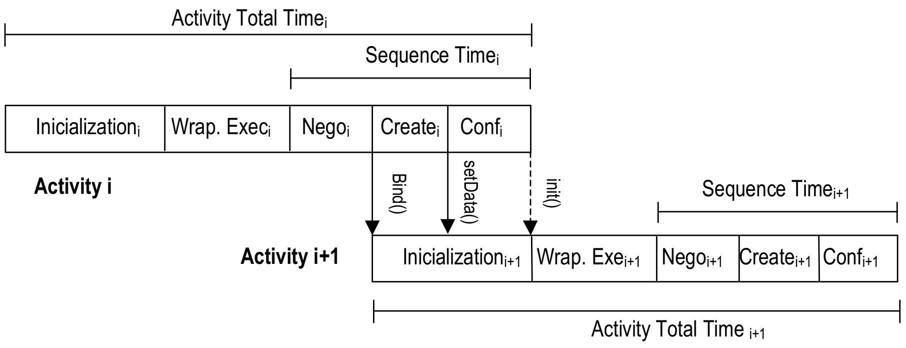

Fig. 8. Activity times collected during tests.

compare the costs associated to the activation, deactivation and migration of the activities.

The activity execution times were collected and computed by the case coordinator. The time measures, described in Fig. 8, represent all the phases of the life cycle of an activity. It is important to highlight that the negotiation, creation and configuration phases require the exchange of messages between consecutive activities. The normalized results of the average times collected in this test are presented in Fig. 9. The first three lines represent the distributed tests, while the remaining represent single host tests.

In Fig. 9, the average activity execution times were normalized and presented together. The first three rows represent distributed execution of cases using two hosts, whereas the last ones represent centralized execution times for each one of the hosts used in the tests.

Note that in these tests, no wrapper is being executed. In a real world application, performed by humans, this time tends to least minutes, hours or even days. The figure allows us to draw two main conclusions about the activity execution costs:

- Casting aside the hardware performance differences, there is no expressive difference between the centralized and distributed relative proportions in the execution times. This can be explained by the use of the same communication mechanism (IIOP over Sockets) implemented by OrbixWeb, in both local and remote server communications. The chart also reveals that the network latency is not very expressive in the overall execution time.
- The time intervals spent in message exchange operations, negotiation and configuration do not represent more than 20% of the total activity time. The biggest latency is associated to the CORBA objects creation, specially the loading of independent Java virtual machines that execute each one of the CORBA objects involved in these tests.

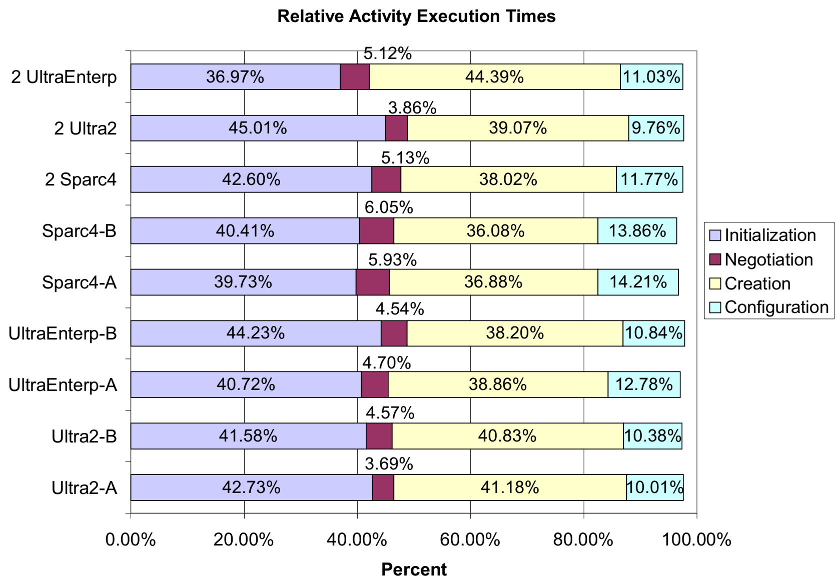

Fig. 9. Relative comparison among average distributed and centralized execution times. Single case execution.

## 5.2. *Scalability tests*

The objective of these tests is to analyze how well the architecture scales when parameters such as number of concurrent cases, size of data generated in the process instance and the activity load are increased.

In order to assess these parameters, we devised three experiments. In the first two tests, the number of concurrent cases simultaneously executing in the system was gradually increased. These tests make use of three different distribution configurations of the WONDER objects: in a single host, in two hosts or in four hosts. In the case of two hosts, the next activity of each process would be executed by an Activity Manager located in the other host, and the coordinators were randomly distributed among the two hosts. In the four hosts configuration, all coordinator components are located in a single host, and the activity managers of each case are randomly distributed amount the remaining three hosts. In the test presented in Fig. 11, the activities did not invoke external applications; in the second case (Fig. 12), a highly CPU consuming operation (bubble sort) was executed during each case step. Finally, the last test (Fig. 13) analyzes the influence of traffic of data in the execution times of the cases.

In the first two tests, the concurrent cases were gradually initiated as described in Fig. 10. In each one of the following tests, 20 concurrent cases, having 20 activities each, were executed. The number of concurrent cases was incremented by 5 at each test round. A delay after each case start was specified in order to avoid a sudden overload of the system.

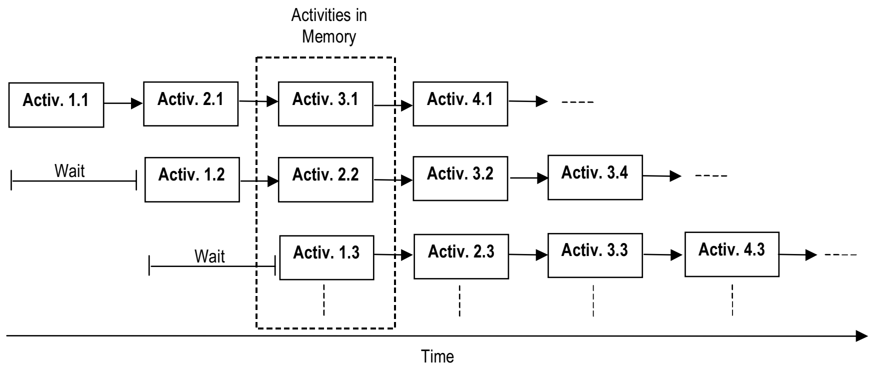

Fig. 10. Initiation procedure of parallel activities.

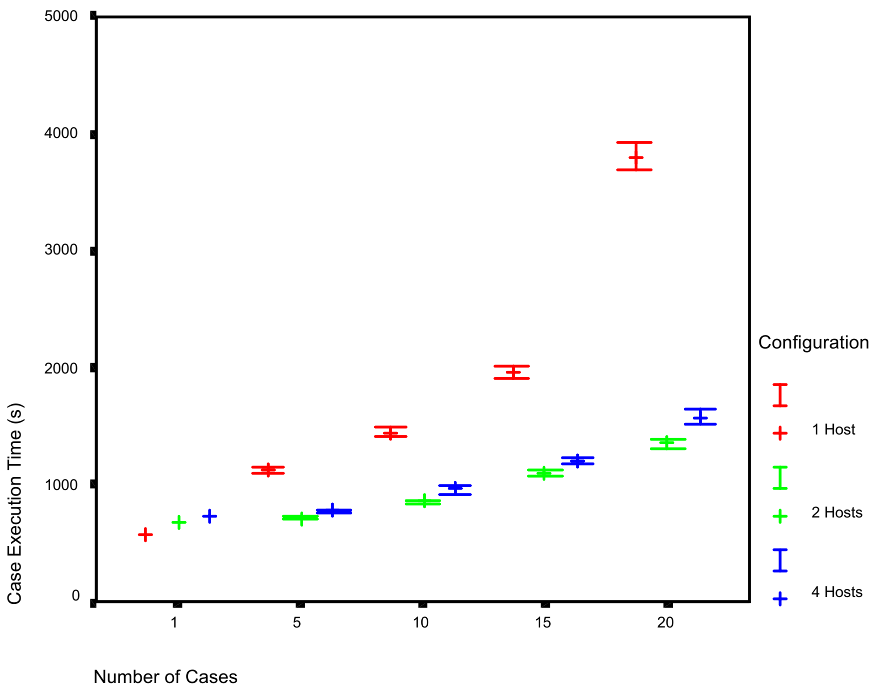

Fig. 11. Average case execution time. No wrapper execution. 1–20 concurrent cases. Error bars refer to a 95% confidence interval.

Figure 11 presents the results collected during the execution of a case whose activities are not CPU intensive. The graph indicates that the degradation in performance for a single host configuration seems to be worse than linear, which would indicate a scalability limitation of such configuration. The data for both the 2 and 4 hosts configurations are consistent with a linear degradation of performance with the increase of the number of concurrent cases, and curiously the 2 host

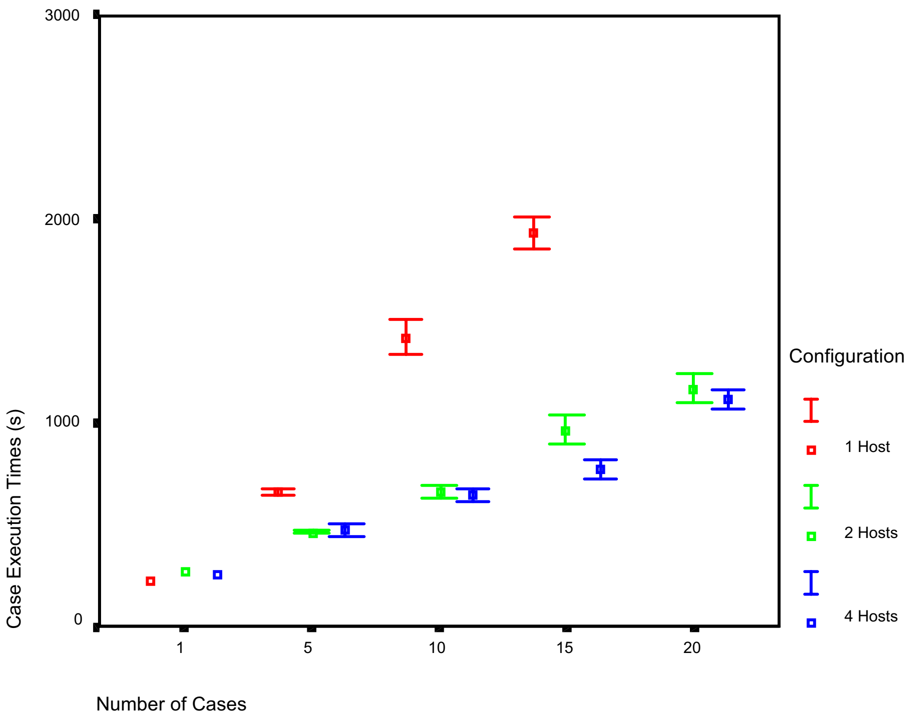

Fig. 12. Average case execution time. Bubble sort of 1000 random numbers. 1–20 concurrent cases. Error bars represent a 95% confidence interval.

configuration has a statistically significant marginally better performance (probably because two of the hosts in this configuration were less powerful machines than the other two, and one of them was assigned to execute some of the activity manager component). In particular, the 2 host configuration has a slope between 39 to 47 (with 95% of confidence) whereas the 4 host configuration has a slope between 49 to 60 extra seconds in the execution of a case for each new concurrent case (a single case in a single host would execute in 588 seconds).

In Fig. 12, in the presence of heavy processing activities, the CPU intensive activities (bubble sort of 1000 random numbers) is shown to impose a more severe delay in the execution times of concurrent activities, to the point that 20 concurrent cases were not able to execute in the single host configuration. In this example, there is no statistically significant difference between the slope of the 2 and 4 hosts configuration. Within a 95% confidence interval, the first is between 42 and 54 and the second between 39 and 49 extra seconds of the total case execution time, for each new concurrent case (a single case in a single host would execute in 225 seconds).

One initial assumption in our model was that the overall system performance would be jeopardized by the increase of data traffic through the network, as the agents and data packages migrate through the many nodes in the system. In order

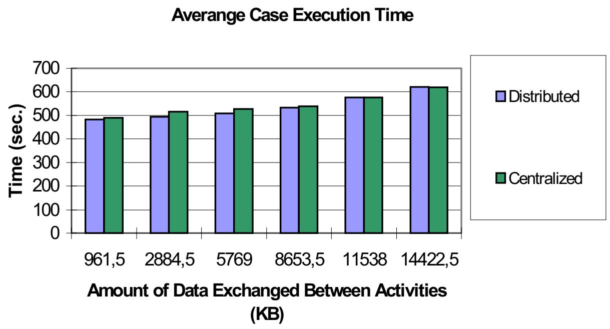

Fig. 13. Average case execution time. Increase of data exchanged among consecutive activities. Single case.

to evaluate this impact, successive tests with increasing case data were performed and compared.

The graphic in Fig. 13 shows that the increase of the case data, exchanged between consecutive activities, did not significantly affect the performance of the system, if compared to the parameters tested in the previous two tests. A ten-fold increase in the size of the data resulted in a 20% increase in the average execution time of the case.

## 5.3. *Tests conclusions*

The ability to distribute the activities among different hosts at runtime, significantly contributes to the capacity of the system to handle an increasing number of concurrent cases, thus, coping with scalability and load balancing. The examples show that distributing the components from a single host to two hosts improves significantly the scalability of the architecture (which in one case goes from a non-linear increase in case execution time to a linear one). The examples were not stressful enough so that limits of the two host configuration were reached and thus we could not show the point in which the four host configuration would improve scalability. The tests also show that there is a linear increase of the case execution time with the increase of data exchanged by the activities.

### 5.3.1. *The influence of notifications*

During the tests, there were not significant delays associated with the asynchronous notifications, sent by the activities to the case coordinators. The same behavior is observed in the transmission of notifications from the case coordinator to the process coordinator. This indicates that the scalability of the system can be increased with the addition of new hosts and the distribution of the activities between them.

Note that nor the case coordinator nor the process coordinators represent bottlenecks since there is one case coordinator per workflow execution instance and process coordinators can be replicated.

### 5.3.2. *Prototype versus full implementation*

Even though the tests were performed using a simplified prototype implementation, the main aspect of the architecture, which is its ability to distribute the activity load in a decentralized fashion, could be analyzed. We argue that the system behavior would not be very different if the other components were fully implemented since: (1) the notifications sent to the coordinators are asynchronous. (2) The processing of these messages does not introduce delays in the agent migration procedure. (3) The backup managers would only execute during low usage periods of the system.

The only set of components that could introduce delays in the agent migration procedure would be the role coordinators. More complex queries, requesting history information could delay the negotiation phase of the agent in some seconds. However, the processing of such queries would be executed in dedicated machines, hosting these components, isolating its impact from the other concurrent activities.

### 5.3.3. *Java and CORBA issues*

The average execution time associated with the centralized execution of the tests overcame the distributed execution after the first increase in the number of concurrent cases. Hence, the use of CORBA objects written in Java, executing in different virtual machines, do not have a good performance in centralized environments, when the number of concurrent cases is big enough to overload this machine. In distributed scenarios, however, where the number of servers executing in one node is smaller, its performance is acceptable, specially if considered that most of the tasks are supposed to be performed by humans in their workstations, which time to accomplish the activities, can take from minutes to days.

The biggest delay, associated to the mobility of the architecture agent is the creation of these objects. This procedure consumes memory and CPU, influencing the performance of the other objects executing in the same host.

This overload of the centralized execution is explained by the way the OrbixWeb manages CORBA objects. It does not differentiate local and remote method invocations. Hence, the method invocations between client and server objects are performed using the IIOP over the TCP/IP stack whether these objects are local or remote. The WONDER architecture does not implement any local data transfer optimization since this responsibility should be provided by the CORBA middleware implementation.

### 5.3.4. *General considerations: accidental versus fundamental issues*

We consider these later CORBA and Java implementation issues as accidental, i.e. they are dependent on particular CORBA implementation features and are independent of the model itself. The execution of CORBA objects as threads instead of different processes as well as the use of local inter-process communications between local objects (as shared memory, for example) are features available in more up-to-date CORBA implementations, not available at the time the prototype was implemented.

The use of the optimizations described in the last paragraph, however, would only "raise the bar" in our tests, allowing the execution of more concurrent cases in a single host. The fundamental problem of centralized control and the inability to fine-tune load balance between different hosts, would still persist. With a significant increase of the number of concurrent cases, the system would be overloaded if not with tenths, with hundreds of concurrent cases, and the centralized bottleneck and single point of failure would still persist. In this case, the addition of new hosts and the use of a fine-grained distributing policy, as in the distributed execution tests, would evenly reduce the execution and coordination loads of the system, obtaining the same results as described in the tests.

## 6. Related Work

Some of the components of the Exotica project,[10,13,14] developed at IBM Almaden Research Center, have similarities to our proposal. In particular the Exotica/FMQM (Flowmark on Message Queue Manager) architecture is a distributed model for workflows, using a proprietary standard (MQI — Message Queue Interface) of persistent message queues. The case data is bundled in a message that is conveyed from one activity to the other through a fault tolerant message oriented middleware. Nevertheless, the proposal is not very detailed on how to deal with all the other requirements for a WFMS described in this paper.

The OMG Workflow Management Facility[17] implements a workflow framework that satisfies the basic workflow management requirements. This specification is based on the WFMC standards and defines a set of basic objects and interfaces. Because of its generality, this specification was not designed to handle the large-scale workflow specific requirements.

The Mentor Project[24] of the University of Saarland defines a traceable and scalable workflow architecture.

Fault tolerance is achieved by using TP-Monitors and logs. CORBA is used as a communication and integration support for heterogeneous commercial components. Scalability is achieved by replicating the data in backup managers. Similar to our architecture, the data and references to data are exchanged between Task List Managers when the activities are being executed and terminated. A limited first prototype was implemented and future extensions should include support for dynamic change of processes and the rollback of cancelled or incomplete workflows.

Rusinkiewicz *et al.* from the Houston University, developed a workflow model based on INCAs (Information Carriers).[4] This model was developed to support the execution of dynamic workflow processes. The process is executed over autonomous execution units (hosts). In this architecture, the process definition and the workflow data are wrapped in a container called INCA, which migrates through the execution units of the system.

The WONDER architecture extends the INCA concept with the mobile agent paradigm, defining active entities (the ActivityManagers) which interpret the case plan. INCAs are passive entities that execute in active hosts (service providers). On the other hand, the WONDER model defines active entities that execute in passive hosts.

Instead of defining compensation actions for fault tolerance, like the INCAs model, WONDER allows the specification of compensation activities. The INCAs checkpointing police, which stores copies of the agent in the hosts of the system, is also used in WONDER. The auditing, monitoring and dynamic allocation of actors are not addressed by the INCAs model.

Recent projects[19,23] have shown the use of the mobile agents paradigm and CORBA in the enactment of WFMSs. Their study is focused on the use of these technologies to provide interoperability and integration of business processes from different organizations. These studies also highlight the advantage of the model in obtaining load balancing and dynamic reconfiguration of the workflow in case of failures and exception handling or dynamic process change. None of them, however, study the feasibility of this technology to address the requirements of large-scale workflow.

The advantages of fully distributed architectures were previous studied.[7] This work originated from the METEOR project and its CORBA implementation, OrbWork.[9] This model defines Task managers, CORBA components that incorporate schedulers (micro-workflow engines) and can be distributed over different hosts in a distributed system in a similar way to WONDER Activity Managers. These components are programmed with their corresponding sub-workflow parts, and are placed at different hosts. Whenever a task is completed, events are generated which trigger the execution of the following tasks. Checkpoints for failure recovery are defined and data references are transmitted between consecutive tasks. The system execution is monitored by external components (watch dogs) analogous to WONDER case coordinators.

Different than WONDER, OrbWork allocates task managers at their execution hosts at the beginning of the workflow. It does not support dynamic allocation of actors and tasks through the use of mobile agents.

Recently, Atluri *et al.*[21] studied the use of decentralized architectures for inter-organizational workflow enactment. The system uses the concept of self-describing workflows (analogous to the mobile case strategy used in WONDER) that are executed by workflow stubs in different sites. The stubs interact with the local resources in each site, invoking execution agents and enacting the corresponding sub-workflow using the resources of each site. The workflow autonomously migrates through different hosts as the process is being executed achieving control and data flow decentralization. In special cases, the issues related to security and conflict-of-interest among organizations are addressed with the use of a modified version of the Chinese Wall Security Model. In this model, policies prevent competitors and non-authorize users to receive information that would give them competitive advantage toward the other workflow participants.

This work addresses some of the security problems identified in our work, and shows the application of a decentralized model in inter-organizational workflow execution. However, no experimental data (qualitative or quantitative) regarding the use of the system is provided.

## 7. Conclusions

In this paper, we presented WONDER, a distributed architecture for large-scale workflow enactment. The architecture is based on the mobile agent paradigm, where workflow activities execute through different user's workstations. The migration follows a workflow plan interpreted by the agents. This characteristic provides decentralization of control. A set of auxiliary components is defined in order to support the agent in its migration and to address the other WFMS requirements. Such decentralization of control and data allows for the definition, enactment and management of large-scale workflows, providing the necessary scalability, decentralization of control and fault tolerance for these applications. It also allows the implementation of load-balancing strategies and policies.

The WONDER uses CORBA's communication framework as its basic communication and distribution system. The CORBA hides all low-level communication and distribution issues, providing location and access transparences in a standard object-oriented programming framework. The use of CORBA as the support environment for such architecture has problems with the persistence of objects. The standard CORBA references were not designed for applications in which objects can be dynamically deactivated and reactivated, in different host ports, during its lifecycle. The information about where an activity should be created and executed is an important issue in our architecture. An application specific naming space was created using persistent location-dependent object names. Some CORBA services were not used due to simplifications and requirements of our architecture.

A prototype version of the system was implemented and performance tests were executed. The tests show that in a single host configuration the system has a worse than linear increase in the case execution with the increase of the number of concurrent cases, even for as low as 15 concurrent cases. But in a 2- and 4-hosts configuration, the increase in execution time is linear. The costs associated to Java and CORBA are dependent on the way the ORB used for the tests is implemented, representing an accidental issue that can be solved by the use of new and optimized ORB implementations. The tests demonstrated, however, that the central issue of scalability can be solved by the distribution of the many activities of each process instance by different hosts in a system.

The ability to arrange the WONDER components in different configurations (ranging from fully centralized to fully distributed), as demonstrated by the tests, shows the flexibility of the approach and its potential use in different load-balancing and distributed arrangements. This flexibility, however, pays a price. It increases the security vulnerability of the system since copies of important workflow information are deposited in less reliable workstations. This approach also introduces many potential points of failure in the system, proportional to the number of hosts used. The decentralized control of the architecture based on mobile agents and a hierarchical monitoring infrastructure, diminishes this problem. Moreover, the failure of a node affects only the workflow cases whose components were standing on that node.

Future extensions include support for dynamic change of process definitions, the implementation of more sophisticated load-balancing policies, and *ad hoc* workflows. In special, the WONDER distributed and autonomous approach facilitates the *ad hoc* change of the plan during the case execution, since the workflow activities and user allocation is done on demand, at runtime, using the process definition enacted by the mobile object.

The study of approaches to safely store the workflow data, as encryption and access control, is also part of the future work.

## Acknowledgments

The authors would like to thank FAPESP (Process 98/06648-0), CNPq, CAPES, and the Pronex — SAI project — MCT/Finep for their support.

## References

1. G. Alonso, D. Agrawal, A. El Abbadi and C. Mohan, Functionality and limitations of current workflow management systems, IBM Technical Report, IBM, 1997.
2. G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör and M. Kamath, Exotica/FMDC: A persistent message-based architecture for distributed workflow management, *Proc. IFIP WG8.1 Working Conf. Information Systems Development for Decentralized Organizations*, Trondheim, Norway, August, 1995.
3. B. R. Odgers, J. W. Shepherdson and S. G. Thompson. Distributed workflow coordination by proactive software agents, *Proc. Intelligent Workflow and Process Management*, *IJCAI-99* Workshop, August 1999.
4. D. Barbara, S. Mehrotra and M. Rusinkiewicz, INCAs: Managing dynamic workflows in distributed environments, *J. Database Management* **7**, 1, 1996.
5. D. Rus, R. Gray and D. Kotz, Transportable information agents, *Proc. First ACM Int. Conf. Autonomous Agents* (1997) 228–236.
6. G. A. Bolcer and R. N. Taylor, Advanced workflow management technologies, software process — Improvement and practice, June 1998, 125–171.
7. J. Miller, A. Sheth, K. Kochut and X. Wang, CORBA-based runtime architectures for workflow management systems. *J. Database Management*, Special Issue on *Multi-Databases*, **7**, 1 (1996) 17–27.
8. S. Jablonski and C. Bussler, *Workflow Management — Modeling Concepts, Architecture and Implementation* (International Thomson Computer Press, 1996).
9. K. J. Kochut, A. P. Sheth and J. A. Miller, ORBWork: A COBRA-based fully distributed scalable and dynamic workflow enactment service for METEOR, Technical

Report UGA-CS-TR-98-006, Department of Computer Science, University of Georgia, 1998.

10. M. Kamath, G. Alonso, R. Günthör and C. Mohan, Providing high availability in very large workflow management systems, *Proc. Fifth Int. Conf. on Extending Database Technology* (*EDBT'96*), Avignon, France, March 25–29, 1996.
11. L. Ismail and D. Hagimont, A performance evaluation of the mobile agent paradigm, *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 1999, 306–313.
12. M. Merz, B. Liberman and W. Lamersdorf, Using mobile agents to support interorganizational workflow-management, *Int. J. Applied Artificial Intelligence*, **11**, 6, 551ff, September 1997.
13. C. Mohan, D. Agrawal, G. Alonso, A. El Abbadi, R. Güthör and M. Kamath, Exotica: A project on advanced transaction management and workflow systems, *ACM SIGOIS Bulletin*, **16**, 1 August 1995.
14. C. Mohan, G. Alonso, R. Günthör, M. Kamath and B. Reinwald, An overview of the exotica research project on workflow management systems, *Proc. 6th Int. Workshop on High Performance Transaction Systems*, Asilomar, September 1995.
15. N. M. Karnik and A. R. Tripathi, Design issues in mobile-agent programming systems, *IEEE Concurrency*, July–September, 1998.
16. OMG, The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995.
17. OMG, Workflow Management Facility, OMG dtc/99-07-05, July 30, 1999.
18. Open Fusion CORBA Services: www.prismtechnologies.com/products/openfusion/main.htm
19. R. Stewart, D. Rai and S. Dalal, Building large-scale CORBA-based systems, *Component Strategies* **59** (January 1999) 34–44.
20. R. S. Silva Filho, J. Wainer, E. R. M. Madeira and C. Ellis, CORBA-based architecture for large scale workflow, IEEE/IEICE Special Issue on *Autonomous Decentralized Systems of the IEICE Transactions on Communications*, Tokyo, Japan, **E83-B**, 5 (May 2000) 988–998.
21. V. Atluri , S. A. Chun and P. Mazzoleni, A Chinese wall security model for decentralized workflow systems, *Proc. 8th ACM Conf. Computer and Communications Security*, Philadelphia, PA, USA, November 05-08, 2001.
22. W. Bausch, C. Pautasso, R. Schaeppi and G. Alonso, BioOpera: Cluster-aware computing, *Proc. 4th IEEE Int. Conf. on Cluster Computing*, Chicago, USA, September 2002.
23. S. Weather, S. Shrivastava and F. Ranno, CORBA compliant transactional workflow system for internet applications, *Proc. Middleware'1998*, 3–17
24. J. Weissenfels, D. Wodtke, G. Weikum and A. Dittrich, The mentor architecture for enterprise-wide workflow management, University of Saarland, Department of Computer Science, 1997.
25. WFMC, The Workflow Reference Model Version 1.1, WFMC-TC-1003, January 1995, http://www.wfmc.org/standards/docs/tc003v11.pdf
26. WFMC, Workflow Standard — Interoperability Wf-XML Binding, WFMC-TC-1023, Version 1.0, May 2000, http://www.wfmc.org/standards/docs/Wf-XML-1.0.pdf
27. WFMC, Workflow Terminology & Glossary Version 3.0, WFMC-TC-1011, February 1999, http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf