A Highly Decentralized Architecture for Large Scale Workflow Enactment

Roberto Silveira Silva Filho, Jacques Wainer, Edmundo R. M. Madeira
IC -Institute of Computing
UNICAMP – University of Campinas
13083-970 Campinas - SP – Brazil
{robsilfi, wainer, edmundo}@ic.unicamp.br

Clarence Ellis
Department of Computer Science
University of Colorado, Boulder, CO 80309
Skip@colorado.edu

Abstract

Standard client-server workflow management systems have an intrinsic scalability limitation, their centralized architecture that hinders the scalability of the system. The whole system is usually executed in a central server that usually represents a single point of failure, which compromises the availability of the system. We propose a fully distributed architecture for workflow management systems. It is based on the idea that the case (an instance of the process) migrates from host to host, executing the case activities, following a process plan. This approach is implemented in the form of a mobile agent. This basic architecture is improved with the addition of other components so that other requirements for Workflow Management Systems, besides scalability, are also addressed. A CORBA-based implementation of such architecture is discussed, with its limitations, advantages and project decisions described. We conclude presenting some performance tests showing the feasibility of such approach.

Keywords*: Large-scale workflow management systems, distributed software architectures, CORBA, distributed systems, 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 systems control, 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 the their execution [4].

Many academic prototypes and commercial WFMSs are based on the standard client-server architecture defined by the WFMC (Workflow Management Coalition) [18]. 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). Workflow activities are executed and coordinated under the command of this server.

This client-server centralized architecture imposes a limiting scalability barrier for the execution of large-scale workflow applications, with many instances of process being executed concurrently. Furthermore, the use of a central database in these systems creates a performance bottleneck and a single point of failure that can paralyze the whole system and, possibly, the whole business itself. Therefore, WFMSs based on centralized client-server architectures are limited in providing appropriate levels of scalability, fault tolerance and availability, which may limit their use on important sets of applications [1].

In this paper we introduce the WONDER (Workflow ON Distributed EnviRonment) architecture, a WFMS architecture based on the mobile agent paradigm. In the WONDER architecture, the control, the storage of data, and the execution of the activities are all distributed over the many hosts of an enterprise computer network. In this paper, we argue that a totally distributed architecture based on mobile agents paradigm can provide the scalability, flexibility and availability necessary for large-scale WFMSs. Other requirements of WFMSs, such as failure recovery, auditing, monitoring and dynamic allocation of users can also be accomplished with this model.

1.1. Scenario

To illustrate our approach, consider the following simplified example. ABC mortgage is a leader mortgage company in Brazil. It has branches operating in many states which are responsible for analyzing and approving thousands of finance applications a day. 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, informing the loan amount requested, the client information and the purpose of the loan. Each request is analyzed separately, in a different loan application process. Once received, the form is routed to an annalist, in the credit department, that checks the client's credit history. This usually requires the access of different credit companies databases. Once approved, the request is forwarded to another analyst, in the finance department, which 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 criteria, an adjusted proposal can be issued, with a reduced credit amount. In both cases, a final proposal is then issued to the client which can accept or reject the proposal. If the propose 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. The workflow for this scenario is represented in Figure 1 as follows.

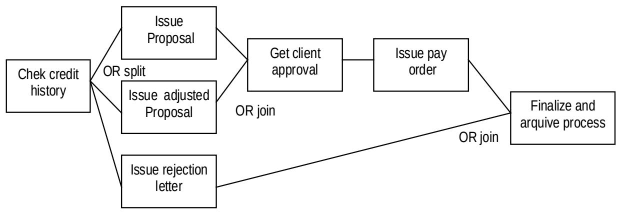

Figure 1. Loan approval process description

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 [18]. 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 Figure 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 or OR branches). **Role** is the generic description of a set of abilities required to a person 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 to where all her work related notifications and activities are send. In particular, the notifications are sent to her **task list**.

1.3. Requirements for Workflow Systems

The main objective of this paper is to present and analyze the characteristics of a distributed architecture that address the scalability issues of large-scale WFMS, hence, the WONDER architecture focuses on addressing the following requirements:

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 get unavailable/unreachable for long periods of time, especially due to failures and use overload.

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

Traceability: History (trace) information of current executing as well as terminated cases must be provided by the WFMS.

Interoperability. Different WFMS should be able to inter-operate, in special, among inter-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.

1.4. Paper Description

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

2. The Distributed Model

Using general terms, the WONDER architecture is based on the idea that each case is represented as a set of mobile agents 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 case moves to a particular user's host once it determines 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.

Some additional components were defined in order to deal with further 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 a credit history checking activity example, the plan will state that a “credit evaluator”, and not a specific actor, should perform the activity of “credit history checking”. In order to cope with this requirement, a role coordinator component, containing information of each role, was defined. In the example above, the case queries the credit evaluator coordinator, and asks it about a user to perform that activity. Once identified the user, 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 a case coordinator component, that keeps track of the 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 the case.

Another important problem for the mobile agent architecture is failure recovery. The distributed characteristic of our architecture introduces many failure-candidate points, but keeps the failure isolated from other processes. What happens to the case if the host where it is executing crashes? To deal with it, some caching policies were specified. For the eventuality of a crash, while the 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 server.

In its essence, the WONDER architecture is structured as a distributed hierarchy of monitoring and policy enforcement components that support the migration and execution of the workflow activities represented as mobile agents. These components correspond to the case coordinator, the role coordinator, the backup server and others described in the next section.

On one hand, our current approach of decentralized components eliminates the bottleneck of traditional workflow systems. On the other hand, distribution is known to increase communication among the decentralized servers, a problem that must be investigated in detail. For example, a 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 server may receive large amounts of data, but this transfer is done asynchronously when network and server load allows for it. The only standard servers, in a client-server sense, are the role coordinators (there is one for each role), which receive a query and must return an answer before the agent migration proceeds. However, the respective amount of information exchanged is also 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.

2.1. Main Components of the Architecture

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

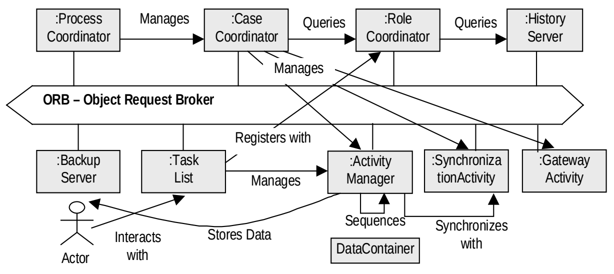

Figure 2. The main components of the architecture

2.1.1. Process Coordinator. The process coordinator is responsible for the creation and management of case coordinators that manage instances of a particular process. Upon a request for a new loan approval, for instance, the "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 managing this loan application activities.

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

2.1.2. Case coordinator. The case coordinator tracks and manages the execution of a particular process instance. It is responsible for detecting activity failures and for coordinating their recovery procedures. It executes the finalization process of a case, collecting cache information left by activity managers in the hosts of the system, and stores the collected data in the history server. It 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.

The 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 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 Coordinator. The role coordinator is a resource locator component. It is responsible for the identification of users qualified to perform a particular activity. It also periodically collects information about the current user status, such as the activities that she is currently executing. 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.

The role coordinator may also have access to the History Server (which stores 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 most experience in that kind of loan?" or "Who was the analyst that approved that line of credit?".

In summary, the role coordinator is the component 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 Activity. 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 different branches of the same case. The synchronization activity 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 Figure 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.

2.1.5. Task List. The user interface is implemented as a task list, similar to an e-mail reader. The 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 task list is the user's 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.

2.1.6. History Server. The history server (or servers) is a front-end 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 Server. The backup server (or servers) is(are) a front-end(s) for the repository of the intermediary state of the active cases. As we mentioned before, a copy of the mobile agent execution state and the workflow data is 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 server 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 servers in the systems, one per process, one for a group of processes, or many for a single process. The identity of the backup server and the moment to perform the backup is parameterized in the case coordinator. This decision is based on many parameters such as network and server loads. Once the backup is made, the user host can erase the past state information of that case.

2.1.8. Activity Manager. This component implements the 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 [9]: there is no mobility of 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 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 created activity manager and the process is repeated until the end of the case.

2.1.9. Application Wrappers. An application wrapper is a component that controls the execution of a particular invoked application. 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 and proceeds in its execution.

2.1.10. Gateway Activity. In order to address the WFMC 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.

3. CORBA Implementation

CORBA [11] was chosen as the middleware to support the WONDER architecture implementation. It provides a set of services and communication transparencies that improve the distributed applications development. CORBA 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 all 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 [13].

3.1. References to CORBA Objects

The CORBA 2.0 standard IORs (Interoperable Object References) is not fully adequate for our application. IORs uniquelly 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 architecture, the CORBA implementation used did not provide such service, hence, our own persistency service had to be developed. In our scheme, the 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 servers, and so on.

In order to provide transparent object persistence, each host has a 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 [12][16]. 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 [12][15]. This approach creates a fail-safe data transfer protocol among different activities, providing a set of “transactional communication channels”.

Large systems require transactional semantics, but may not always require distributed transactions [14]. In the WONDER architecture the Activity Manager peers manage the consistence of the data transfer. 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 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 the fail 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 [Web-1]. 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 de-coupled by the notification service?

The WONDER architecture does not rely on any standard CORBA naming service because of the IOR problems described in section 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, which specialization implements the LOA object.

3.3. Execution Scenarios

In this section, some execution examples are presented. They emphasize on 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.

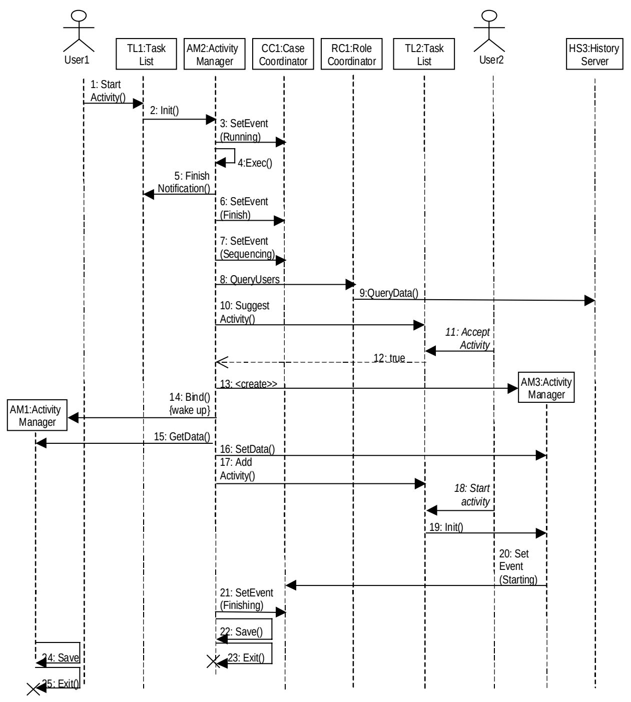

Figure 3. Activity sequencing diagram

3.3.1. Activity Sequencing. The 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 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 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.

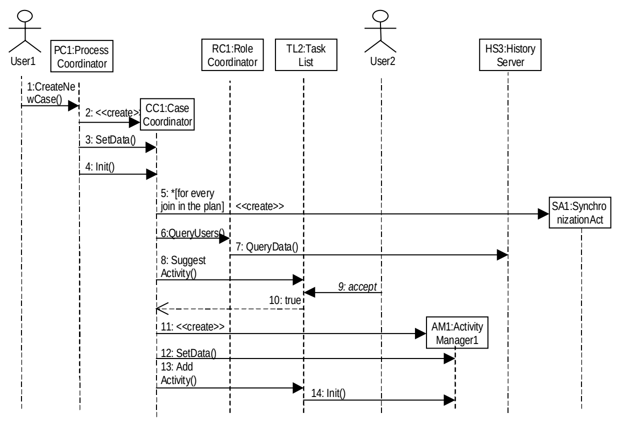

Figure 4. Case creation sequence diagram

3.3.2 Case Creation. The case creation procedure, presented in Figure 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 Figure 3 is iterated for each activity in the branch. The new 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.

The synchronization procedure involving the activities AM1, AM2, and SA4 is described in Figure 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 3.3.1. As usual, CC1 is kept informed of the progress of the case, managing the case and handling its failures.

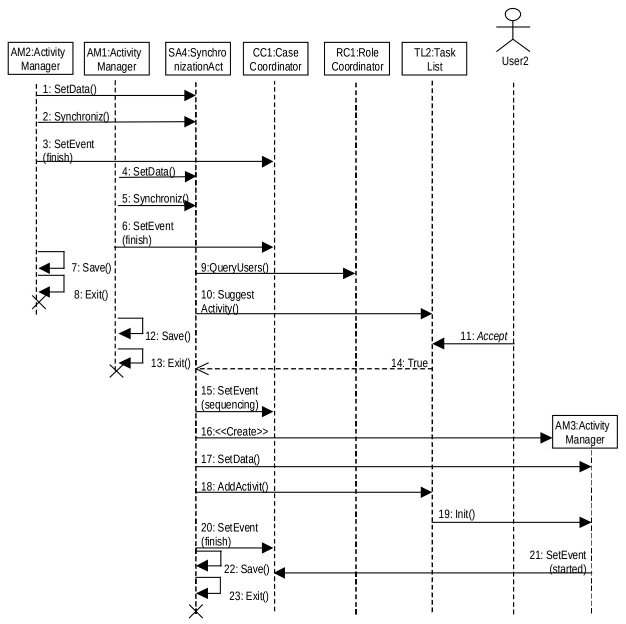

Figure 5. An and-join synchronization diagram

3.3.5. Case Finalization. The Figure 6 presents the sequence diagram of a case finalization procedure. By the end of each case, data stored at each host that executed at least one activity of the case, and all case data stored in the backup server(s) are removed by the case coordinator CC3 (9, 11 and 13). An execution summary containing relevant data for future queries is stored in the History Server HS2 (12).

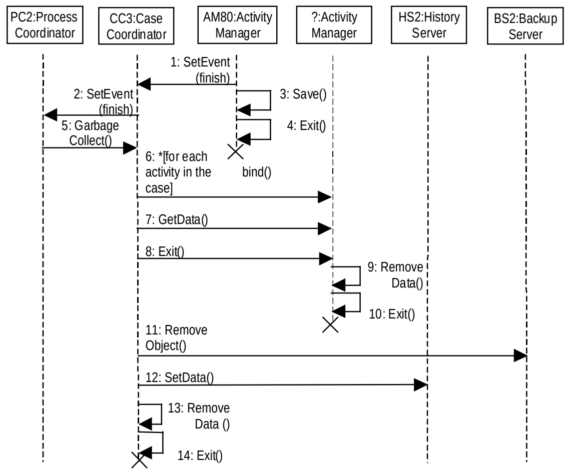

Figure 6. A sequence diagram of a finalizing case

3.3.6. Failure Recovery. The failure recovery process 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. This routine is managed by the Case Coordinator, using data stored in the object repository of each host, and in the backup servers scattered over the system.

4. Performance Tests

This section describes the performance tests executed to validate our approach. The tests were designed to analyze the feasibility and behavior of the architecture components in different distributed and centralized configurations.

For these tests, the core components of the WONDER 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 JDK1.1), using the Iona OrbixWeb 3.1c, 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.

4.1. Overhead Tests

In order to determine the influence of the WONDER runtime environment 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 Figure 7 were collected. Two comparative charts with the results are presented in Figure 8 and Figure 9. The test consisted in the execution of a single case, with 20 consecutive activities in two different configurations, centralized and distributed, for every pair of machines used in the tests. In the centralized tests the activities of the test case and the coordinators execute in a single 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.

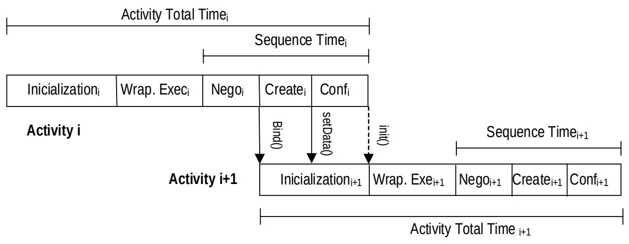

Figure 7. Activity times collected during tests.

The activity execution times were computed by the case coordinator. The time measures, described in Figure 7, represent all the phases of the life cycle of an activity. This information was collected by the case coordinators in distributed and centralized scenarios. The results are presented side-by-side in Figure 8 for comparison.

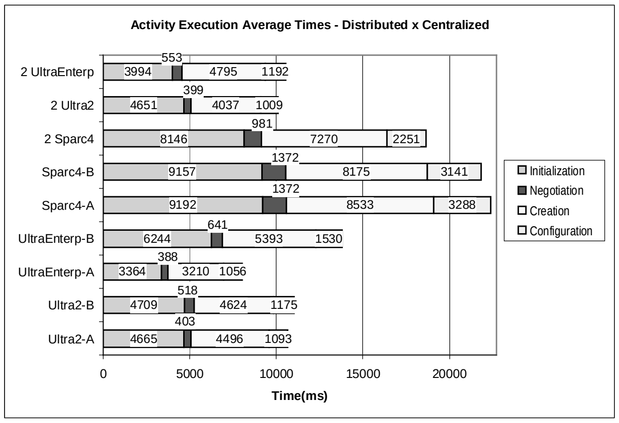

Figure 8. Comparison among average distributed and centralized activity execution times. Execution of a single case.

In general, the distributed execution was faster than the centralized one in machines with the slowest and medium processing power machines as the SPARC-Station4 and Ultra2 pairs. In these hosts, the addition of a new concurrent activities has a more prominent influence in the overall performance of the system.

The better performance of the distributed executions is also a consequence of the execution of the case coordinators in a third machine. During the tests performed in centralized environments, the coordinators were executed in the same host as the activities, which contributes to the loading of the whole system. This influence is more expressive as the processing capacity of the host is lower.

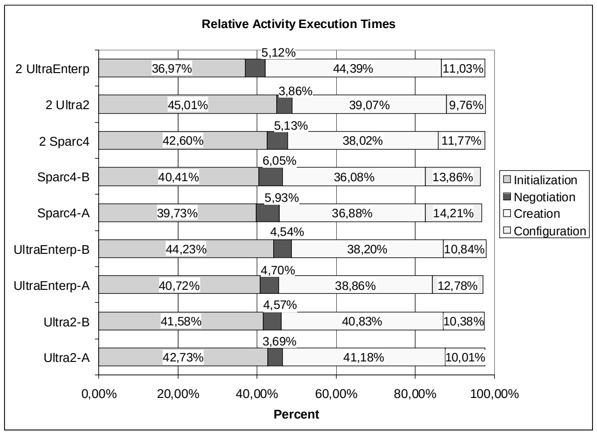

Figure 9. Relative comparison among average distributed and centralized execution times. Single case execution

The Figure 8 allows the comparison of the relative proportions of the sequence times. As shown in the graph, these relative values are very similar in the tests. The relative differences between local and remote invocations are not expressive. 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.

One can argue that the 23 seconds average migration and initialization time, measured for SPARCStation 4 hosts, the slowest machines, is an expensive price to pay. This overhead is relatively big for runtime applications, for example. However, for conventional business applications, which activities can elapse minutes or even hours, this migration latency is acceptable.

It can also be observed that 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.

4.2. Scalability Tests

The objective of these tests is to analyze the behavior of the system when multiple cases are being executed in parallel on distributed and centralized scenarios. We measure the average execution times of the activities and the whole case, in three different conditions. In the first set of tests (Figure 11), the activities did not invoke external applications; in the second case (

Figure 12), a highly CPU consuming operation (bubble sort) was executed during each case step. Finally, the last test analyzes the influence of the case data in the execution times of the cases. This last test (Figure 13) had the objective of exploiting the use of the workflow for activities in which a large amount of data needs to be transferred between activities.

.

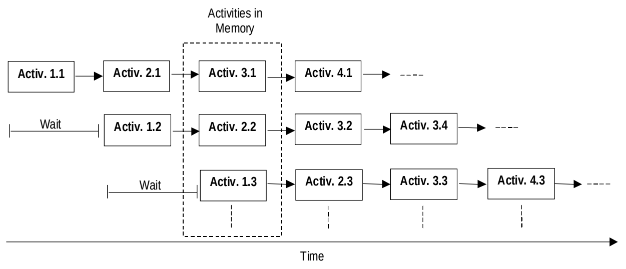

Figure 10 Parallel activities initiation procedure

The concurrent cases were initiated as described in Figure 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.

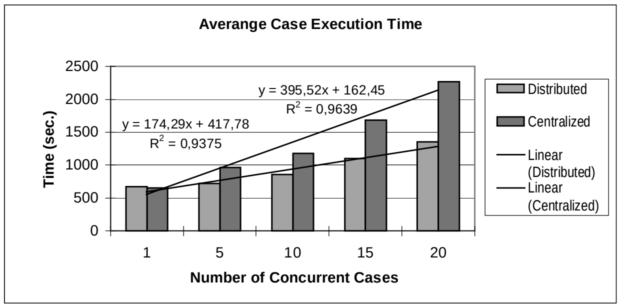

Figure 11. Average case execution time. No wrapper execution. 1-20 concurrent cases.

In Figure 11, the distributed configuration performs better than the distributed one. This difference becomes more expressive as the number of concurrent cases increase. The delays associated with the case execution have a linear growth with the increase of number of concurrent cases.

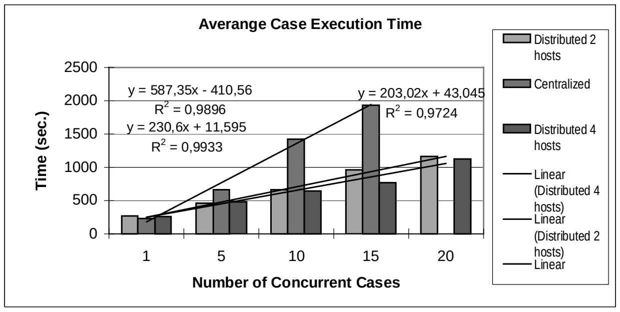

Figure 12. Average case execution time. Bubble sort of 1000 random numbers. 1-20 concurrent cases

In

Figure 12, in the presence of heavy processing activities, the system presented the same linear behavior. The more hosts are added in the distribution set of servers, the lower the increase (angular coefficient of the approximation line) of the case execution delay.

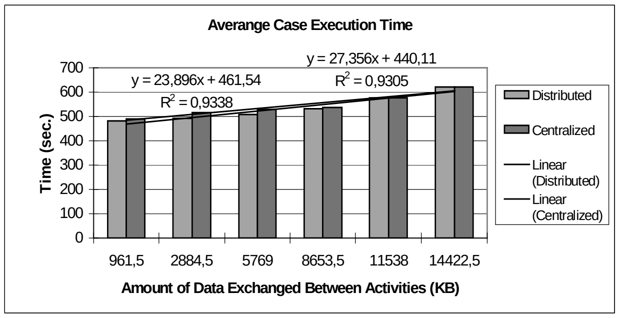

Figure 13. Average case execution time. Increase of data exchanged among consecutive activities. Single case.

In

Figure 13, the increase of the case data, exchanged between consecutive activities did not significantly affect the performance of the system, as in the previous two tests. A ten fold increase in the size of the data resulted in 20% increase in the average execution time of the case.

4.3. Tests Conclusions

The architecture presents a linear increment of the average case execution time with the increase of the number of concurrent cases, the volume of data exchanged between consecutive activities, and the work load of the wrappers executed. The angular coefficient of the approximation lines (the delay increase rate) in the charts decreases with the increase of the number of hosts used in the tests.

4.3.1 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.

4.3.2 Prototype versus full implementation

Even though the tests were performed using a simplified prototype implementation, its behavior would not be very different if the other components were fully implemented. The notification sent to the coordinators are asynchronous. The processing of these messages do not introduce delays in the agent migration procedure. Backup servers would only execute during low usage periods of the system.

The only component that could introduce delays in the agent migration procedure is the Role Coordinator. More complex queries, requesting history information could delay the negotiation phase of the agent in some seconds. This query is specific of some activities and is out of the scope of the present work.

4.3.3 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 objets written in Java, executing in different virtual machines, do not have a good performance in centralized environments, where the number of concurrent cases is big. In distributed scenarios, however, where the number of servers executing in one node is smaller, its performance is acceptable.

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 is 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.

4.3.4 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 issues 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.

The use of the optimizations described in the last paragraph would only “raise the bar” in or tests, allowing the execution of more concurrent cases in a single host. The fundamental problem of centralized execution, however, 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 would still persist. In this case, the addition of new hosts, as in the distributed execution tests, would be used to distribute the case execution and obtain the same results as described in Figures 11 to 13.

5. Related Work

Some of the components of the Exotica project [5,6,7,8], 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 queues. The case data is bundled in a message that is conveyed from one activity to the other through a fault tolerant persistent queue. Nevertheless, the proposal is not very detailed on how to deal with all the other requirements for a WFMS.

The OMG Workflow Management Facility [10] 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 [11] of the University of Saarland is a scalable, traceable 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 servers. 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) [14]. 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.

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 which 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, the 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 the WONDER model. The auditing, monitoring and dynamic allocation of actors is not addressed by the INCAs model.

Recent projects [15] and [16] 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.

6. Conclusions

In this paper, we present the WONDER, a distributed architecture for large-scale workflow enactment. The architecture is based on the idea that the case moves through different workstations to the user’s actual host. The migration follows a workflow plan which provides autonomy to the activities. The case is implemented as a mobile agent, in which there is no code mobility. A set of coordinators and servers were added to the basic architecture so that all other requirements of a WFMS could also be addressed. 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.

The WONDER uses the CORBA 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. Some performance tests were executed. Because we could not compare the WONDER approach with other centralized systems, the feasibility of the model was studied instead. The architecture was compared with itself in different arrangements, showing that the system was best suited for distributed execution in terms of performance.

The tests demonstrate that the delays associated with the workflow execution are minimized as new hosts are used by the system. The more hosts are used in the system, the better its execution performance. In an ideal case, in which each user has his own host and not many simultaneous activities are being performed by each user, the system provides adequate scalability.

In these tests, the system had a linear increase of the execution times with the addition of new concurrent cases. 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 fundamental problem 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 (centralized or 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 is deposited in less reliable workstations. This approach also introduces many potential points of failure in the system, in a way proportional to the number of hosts used. These occurrence of a failure, however, if compared to a centralized approach, do not make the whole system unavailable since the fault of a host affects only the cases that have activities in execution in that host, not hindering the whole system execution.

Future extensions include support for dynamic change of process definitions, and *ad-hoc* workflows. The WONDER distributed and autonomous approach facilitates the 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 first three authors would like to thank FAPESP (Process 98/06648-0), CNPq, CAPES, and the Pronex - SAI project - MCT/Finep for their support.

7. References

[1] Alonso G., Agrawal D., El Abbadi A., Mohan C. "Functionality and Limitations of Current Workflow Management Systems". *IBM Technical Report*, IBM, 1997.

[2] Alonso G., Agrawal D., El Abbadi A., Mohan C., Günthör R., Kamath M.. Exotica/FMDC: A Persistent Message-Based Architecture for Distributed Workflow Management. Proceedings of the IFIP WG8.1 Working Conference on Information Systems Development for Decentralized Organizations, Trondheim, Norway, August, 1995.

[3] Barbara D., Mehrotra S., and Rusinkiewicz M. INCAs: Managing Dynamic Workflows in Distributed Environments. Journal of Database Management, Vol. 7, No. 1, 1996.

[4] Jablonski S., Bussler C. Workflow Management - Modeling Concepts, Architecture and Implementation. International Thomson Computer Press, 1996.

[5] Kamath M., Alonso G., Günthör R., Mohan C.. Providing High Availability in Very Large Workflow Management Systems. In Proceedings of the Fifth International Conference on Extending Database Technology (EDBT'96), Avignon, France, March 25-29,1996.

[6] Merz M., Liberman B., Lamersdorf W.Using Mobile Agents to Support Interorganizational Workflow-Management. International Journal on Applied Artificial Intelligence, 11(6), S. 551ff. September 1997.

[7] Mohan C., Agrawal D., Alonso G., El Abbadi A., Güthör R., Kamath M.. Exotica: A project on Advanced Transaction Management and Workflow Systems. ACM SIGOIS Bulletin, Vol. 16, No. 1, August, 1995.

[8] Mohan C., Alonso G., Günthör R., Kamath M., Reinwald B. An Overview of the Exotica Research Pro-

ject on Workflow Management Systems. Proc. 6th Int. Workshop on High Performance Transaction *Systems*, Asilomar, Set. 1995.

[9] N. M. Karnik and A. R. Tripathi. Design Issues in Mobile-Agent Programming Systems. IEEE Concurrency, July-September, 1998.

[10] Odgers B.R., Shepherdson J.W. and Thompson S.G. Distributed Workflow Co-ordination by Proactive Software Agents. In Proc. of Intelligent Workflow and Process Management, IJCAI-99 Workshop, August 1999.

[11] *OMG.* The Common Object Request Broker: Architecture and Specification. Revision 2.0 July 1995.

[12] OMG. Workflow Management Facility. OMG dtc/99-07-05, July, 30, 1999.

[13] Silva Filho R. S., Wainer J., Madeira E. R. M., Ellis C. CORBA Based Architecture for Large Scale Workflow. IEEE/IEICE Special Issue on Autonomous Decentralized Systems of the IEICE Transactions on Communications , Tokyo, Japan, Vol. E83-B, pp.988-998. No. 5. May 2000.

[14] Stewart R., Rai D., Dalal S. Building Large-Scale CORBA-Based Systems. Component Strategies pp.34-44, 59, Jan. 1999.

[15] Weather S., Shrivastava S., Ranno F. CORBA Compliant Transactional Workflow System for Internet Applications. In Proc. Middleware'1998, pp. 3-17

[16] Weissenfels J., Wodtke D., Weikum G., Dittrich A. The Mentor Architecture for Enterprise-wide Workflow Management. University of Saarland, Department of Computer Science, 1997.

[17] WFMC. The Workflow Reference Model Version 1.1, WFMC-TC-1003. Jan. 1995. http://www.wfmc.org/standards/docs/tc003v11.pdf

[18] WFMC. Workflow Terminology & Glossary Version 3.0. WFMC-TC-1011, Feb. 1999. http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf

8. Web References

Web-1. Open Fusion CORBA Services:
www.prismtechnologies.com/products/openfusion/main.htm
Web-2. WONDER Project:
/www.dcc.unicamp.br/~931680/wonder