CORBA Based Architecture for Large Scale Workflow

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}@dcc.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, the central server, which represents a bottleneck for large-scale applications. This server is also a single failure point that may disable the whole 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, following a process plan, while the case activities are executed. This basic architecture is improved so that other requirements for Workflow Management Systems, besides scalability, are also contemplated. A CORBA-based implementation of such architecture is discussed, with its limitations, advantages and project decisions described.

Keywords: *Large-scale Workflow, Distributed Objects, CORBA, Distributed Systems, and Mobile Objects.*

1. Introduction

Workflow Management Systems (WFMSs) are used to coordinate and sequence business processes, such as loan approval, insurance reimbursement, and other office procedures. Such processes are represented as workflows, computer interpretable description of activities (or tasks), and their execution order. The workflow also describes the data available and generated by each activity, parallel activities, synchronization points and so on. This description may also express constrains and conditions 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 activity execution [3].

Many academic prototypes and commercial WFMSs are based on the standard client-server architecture defined by the WFMC (Workflow Management Coalition) [2]. In such systems, the Workflow Engine, the core of a WFMS, is executed in a server machine 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).

This client-server centralized architecture represents a limiting barrier for large-scale applications, with many instances of process being executed concurrently. Furthermore, the use of a central database in these systems represents a performance bottleneck and a single failure point 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 hind their use on an important set of applications [4].

In this paper we introduce the WONDER (Workflow ON Distributed EnviRonment) architecture, a WFMS that addresses, in special, the scalability and availability issues. Other requirements of WFMSs, such as failure recovery, auditing and traceability are also addressed. In the WONDER architecture, the control, the storage of data, and the execution of the activities are all distributed over the hosts of an enterprise computer network.

1.1. 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 [2]. A **case** is an instance of a process. Processes are defined in terms of **activities** or **tasks**, which are atomic actions performed by a single person or by a program. **Role** is the generic description of a set of abilities required to a person in order to perform an activity. Thus, secretary, programmer and reviewer 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 send to her **task list**.

1.2. Requirements for Workflow Systems

In this paper, we will address the following requirements of a WFMS:

Scalability: The WFMS should not have its performance degraded due to the increase of: processes, cases or activity instances within a workflow. It should also support a big volume of application data and actors.

Failure recovery: The WFMS should deal with both software and hardware failures with the least intervention of users.

Availability: The WFMS must not get unavailable/unreachable for long periods of time.

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 the current and terminated cases must be provided by the WFMS.

1.3. Paper Description

The next section discusses, at a glance, 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 some implementation issues, Section 5 describes some related work and Section 6 presents some conclusions.

2. The Distributed Model

In general, and using informal terms, our architecture is based on the idea that each case is a "mobile agent" that migrates from host to host as the case activities are performed. The agent encapsulates both, the application data and the plan for that case (workflow control data). The case "moves" to a particular user's host once it "figures out" that the next activity will be performed by that user at that host. Once the activity is finished, the agent "figures out" another user to perform the next activity and moves to his/her host. This "mobile agent" architecture copes with the scalability requirement, since there is no central control or data server, and there is no performance bottleneck.

Some components were added in order to deal with further requirements. It is usual that the plan of a process does not specify a particular user as the performer of an activity, but only a role. Consider a credit checking activity example, the plan will state that a credit evaluator, but not a specific actor, should perform the activity of credit 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 figured out the user, the case moves to that user's host.

Monitoring is also an issue in our architecture. How does one find out, without broadcasting, what is the current state of a case, since it may be executing in any of the hosts of the network? A case coordinator component, that keeps track of the case as it moves along, was defined. Each time the case moves to a new user's host, it sends a notification to its case coordinator. Therefore the case coordinator knows where and at which process stage is a 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 breaks down? To deal with it, some redundancy policies were specified. For the eventuality of a break down, while the case is executing in the current host, a persistent copy of its last state is stored at the previous host. As soon as the failure is detected, the case coordinator elects another host/user to restart or resume the halted activity, using the past case state. Furthermore, to avoid unnecessary storage of old activity states throughout the network, the case coordinator may direct hosts to transfer these persistent state to a backup server, freeing their disk space.

In its essence, the "mobile agent" is composed of an hierarchy of responsibilities, where each server manages a subset of objects. These servers correspond to the case coordinator, the role coordinator, the backup server and others. Our current approach of decentralized servers removes the bottleneck of traditional workflow systems. In counterpart, distribution is known to increase communication among the decentralized servers, a problem that must be investigated in detail. For example, a case coordinator represents one instance of a process and receives very short asynchronous notifications from "mobile agents". 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 server, in a client-server sense, is the role coordinator, which receives a query and must return an answer before the case migration carries on. However, the respective amount of information exchanged is also small, comprising the sending of a short query and the return of a user identity of as an answer.

Therefore, since message exchanging is small and asynchronous, communication is not a problem.

2.1. Main Components of the Architecture

The architecture in Figure 1 is composed of autonomous distributed objects, which are described in the next subsections.

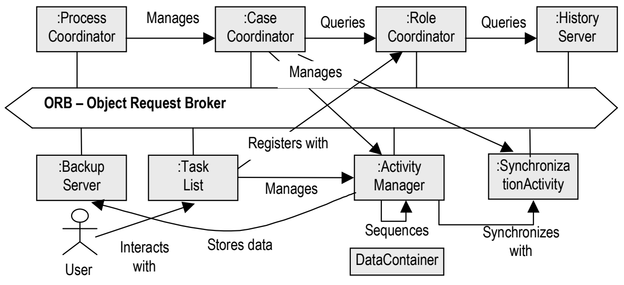

Figure 1. The main components of the architecture

2.1.1. Process Coordinator. The process coordinator is responsible for the creation and management of case coordinators. Upon a request for a new office supply purchase, for instance, the “purchase of office supply process coordinator” will create a new case coordinator for this order, transferring the plan to this new object.

If one needs to locate all instances of a process, the process coordinator also keeps track of all of its case coordinator instances in execution. For example, if the definition of the process is changed, say to introduce a new activity, the process coordinator will propagate these changes to all its cases.

2.1.2. Case coordinator. The case coordinator centralizes all status information of a particular case. It is responsible for detecting case failures and for coordinating its recovery procedures. It executes the finalization process of a case, by performing the garbage collection of past activities data and state, and storing the collected data in the history server. It also answers queries about the current case state, notifies the process coordinator when a case is terminated, as well as other management procedures.

2.1.3. Role Coordinator. The role coordinator is responsible for the management of the users able to perform a particular role. It also manages the current user status, such as the activities that the user is currently executing. With this information, a “programmer role coordinator” can answer queries like “Which is the least loaded programmer?” or “Which are the available programmers?”.

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 programmer with most experience in that kind of system?” or “Who was the programmer that implemented the previous version of that code?”.

2.1.4. Synchronization Activity. And-joins and Or-Joins are a particular problem in "mobile agent" architectures. Each join of a case must be created before its beginning, otherwise a "mobile agent" would not know where to go when it needs to synchronize with other "mobile agents", that are executing in different branches of the same plan. The synchronization activity will wait for all notifications (and-join) or the first notification (or-join) from its input activities before starting the following (output) activity. For example, during an and-join, once all "mobile agents" have moved from its input activities to the synchronization activity, this synchronization object merges all case data, and composes a new single "agent" that is moved to the host assigned to the next activity. In an or-join synchronization activity, the first case to arrive will trigger the sequencing of the next activity. A synchronization activity may also wait for other synchronization signals, such as external events.

2.1.5. Task List. The user interface is implemented as a task list, similar to a mailbox. 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, so it should also allow for some customizations, such as selection of preferred external applications, change of the user's preferential host, selection of policies for sorting the coming activities, and so on. It also collects information about the user's workload, to be queried by the role coordinators.

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, the past state information about a “mobile agent” is stored in some of the hosts where it executed. These users’ hosts are neither trusted to hold the past state information indefinitely, nor to be active when this information is needed. The backup server runs in a more reliable and powerful machine. It receives the data and state of the past activities of an active case, under the command of the its case coordinator. Once the backup is performed, the state information can be erased from the users' hosts.

2.1.8. Activity Manager. So far, we have been using the idea of a “mobile agent” as an intuitive description of the distributed nature of a case. The case, however, is not really implemented as a “mobile agent” in its strict terms, but as data and state that are transferred between two specific objects. There is no code mobility.

Each activity manager coordinates the execution of an instance of an activity for each case. When a new activity of a case needs to be performed, a new activity manager is created at the preferential host of the user that will perform the activity. It is, then, configured with the activity specific data, and the previous activity case state. The plan interpretation is resumed and the activity is performed using the appropriate applications, through the use of application wrappers. The activity manager waits until the user finishes 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 selected user to perform the next activity. After that, it transfers the case information to the created activity manager. It also receives requests from the case coordinator to transfer its case data to a backup server.

2.1.9. Application Wrappers. The application wrappers are objects that control the execution of a particular invoked application. It launches the application with initial data and files and collects 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.

3. CORBA Implementation

The CORBA communication framework [9] provides a set of functionalities and transparencies that improve the distributed applications development. It implements 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, polymorphism and so on. It also enables the use of legacy applications, which were developed for different hardware and software platforms, through the definition of IDL interfaces to these legacy applications.

3.1. References to CORBA Objects

The main problem using CORBA as the support environment for the distributed workflow architecture is its object reference specification. CORBA standard IORs (Interoperable Object References) are not adequate for our application. These references are dynamically allocated, and include the IP address and port number, which respectively locate the host and an object within it. Since the completion of a case may take up to many days, or even months, one cannot assume that, for a whole case execution lifecycle, an object will keep itself active, on the same port it was created, being located by the same IOR. The OMG (Object Management Group) CORBA specification still lacks an object persistence service, therefore we had to create our own persistent object references. In our scheme, the objects are locally stored, and identified using the following naming structure: (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). The LOA executes as a hook in the WONDER runtime environment daemon (orbixd – OrbixWeb locator daemon) and intermediates the object activation (bind), deactivation and persistence, saving the object state and data in a local reserved disk area (object repository). For example, the case coordinator for a request for the purchase of 500 paper clips (case C4375), of the process "purchase of office supply" (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 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 IOR of the newly restored object to be immediately used.

3.2. Hierarchy of Interfaces

We describe below the main aspects of the mapping between the components of the architecture and the CORBA environment. A hierarchy of interfaces (and objects) is described.

The interface hierarchy, presented in Annex A, is composed of three interface groups: *Repository*, *WonderCoordinator* and *Activity* interfaces. *Repository* represents interfaces implemented by objects that manage data storage. There are two kinds of data repositories: *BackupServer* and *HistoryServer*. *Coordinators* manage the execution of other system objects. There are case and process coordinators. *Activity* instances are objects that control the task execution. They are managed by *CaseCo-*

ordinator instances. There are two subclasses of *Activity*: one to perform and control the execution of tasks by people (or programs) (*ActivityManager*), and one to carry out synchronization points (*SynchronizationActivity*). The *ActivityManager* also implements the activity sequencing and execution. The activity sequencing procedure uses *DataContainer* instances to store data and process definitions. These containers are exchanged among *ActivityManager* objects. The coordinator and activity groups, along with the *RoleCoordinator*, are sub-interfaces of the *WorkflowObject*. The *RoleCoordinator* objects manage dynamic and history information concerned with the system users (*User* instances). Each user has an associated role. *WonderObject* instances are uniquely identified, and can be controlled, located and stored. Local Object Activators (*LOA*) are responsible for implementing the objects persistence and activation. *TaskList* instances store information concerned with user allocated activities.

3.3. Execution Scenarios

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

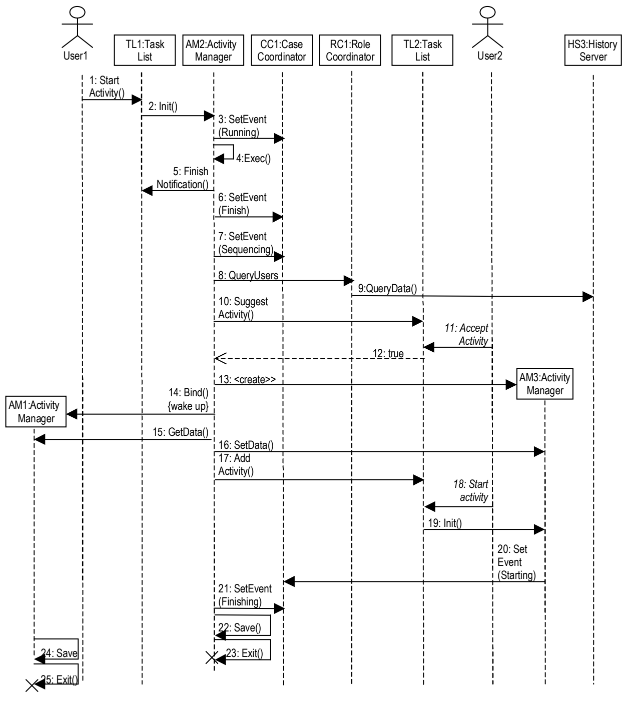

Figure 2. Activity sequencing diagram

3.3.1. Activity Sequencing. Figure 2 shows a typical example of an activity sequencing procedure. 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 “figures-out” which activity is to be performed next, and by which role. The AM2 queries RC1 (the role coordinator for the role to execute the next activity - message 8), which selects a user to perform the next activity. The AM2 places the notification of the new activity in the user’s task list TL2 (10). If the selected user accepts the activity, the activity creation procedure starts (10 to 13). The activity manager AM2 creates 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 data to send to AM3 locally, it gets data 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.

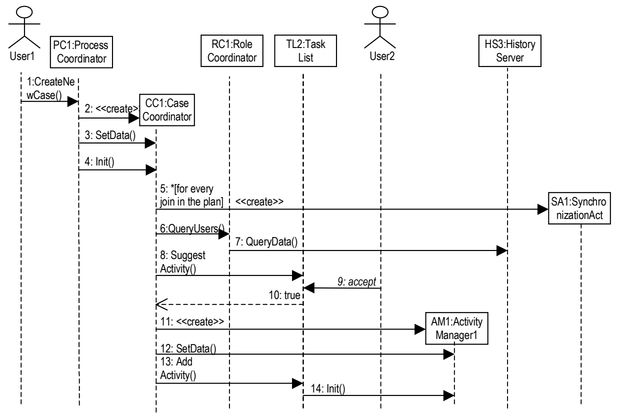

Figure 3. Case creation sequence diagram

3.3.2 Case Creation. The case creation procedure, presented in Figure 3, 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 2 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 refer directly to this synchronization activity address.
The synchronization procedure involving the activities AM1, AM2, and SA4 is described in Figure 4. 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.

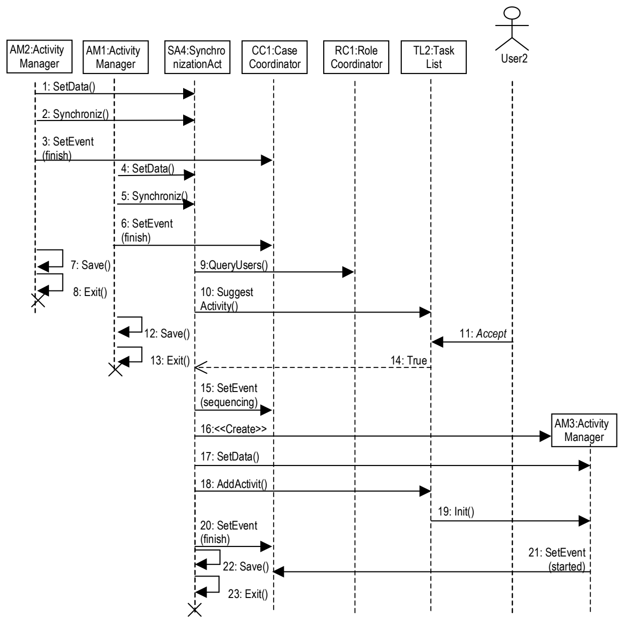

Figure 4. An and-join synchronization diagram

3.3.5. Case Finalization. The Figure 5 presents the 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).

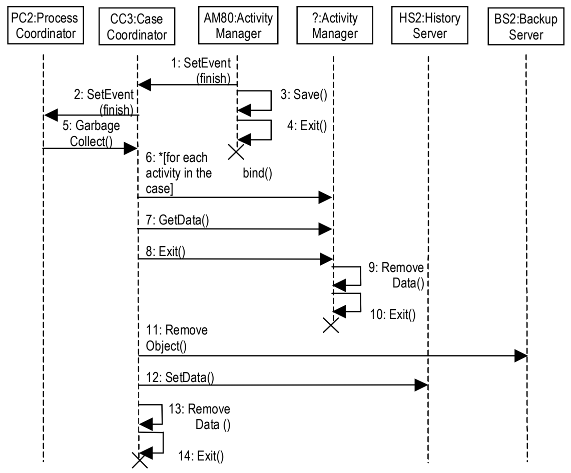

Figure 5. A sequence diagram of a finalizing case

3.3.6. Failure Recovery. The failure recovery process consists of: 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. Implementation Issues

The system was developed in the Institute of Computing at UNICAMP. It was written in Java (Sun JDK1.1), using the Iona OrbixWeb 3.1c, a Java ORB implementation. The distributed system used during the implementation is composed of Unix Workstations, NCD Diskless X-terminals and Windows NT/Linux PC Workstations. All computers are connected by a 10Mb Local Area Network.

4.1. CORBA Services

Many CORBA based Workflow architectures use a subset of the OMA CORBA Services [10,11]. 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 [10,13]. This approach creates a fail-safe data transfer protocol among different activities, implementing a set of "transactional communication channels". Large systems require transactional semantics, but may not always require distributed transactions [12].

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 procedure, creating an alternative path to be followed. This simple approach dispenses a more complex control implemented by a transaction server.

The Event and the Notification Services decouple the producer and consumer servers, 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.

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 a locator that resolves markers (OrbixWeb user-friendly object names) to IOR object references. This locator, operating with the LOA, is also used to implement the objects activation and deactivation, besides their persistence. The locator is implemented using the *OrbixWeb orbixd* daemon and an *OrbixWeb LoaderClass* hook, which specialization implements the LOA object.

4.2. Workflow, CORBA and Java Implementation. Advantages and Drawbacks

The core CORBA mechanism (ORB) implements what could be described as a “statefull remote procedure call”. Objects are created and can remain active in memory, keeping its state, for a long time after their use. The processes of activation and deactivation of such objects are time consuming operations. CORBA objects were not designed to be constantly created and destroyed. Furthermore, in many Java CORBA implementations, one virtual machine is assigned for each object created. This approach creates processes which may overload a typical desktop workstation. Unloading inactive servers from memory solves this problem. This is performed, periodically, by the LOA which saves and kills the inactive objects through a timeout mechanism. Hence, in a typical use case, there will be only one activity manager in main memory, the server corresponding to the current activity in execution.

The activation and deactivation delays, associated with the CORBA objects, however, represent a fraction of the total activity time. Workflow processes are usually enacted by human users. In a typical scenario, electronic forms and data are routed among actors in a company, who can read, fill or create these documents. Such activities may elapse minutes, hours, or even days. Hence, the object activation delays do not represent a problem to our application.

The workflow state mobility is easily implemented in Java. The Plan Interpreter, the core of the Workflow engine, is a Java object that is paused, serialized and carried among adjacent Activity Managers using the Java object serialization facilities. The serialized object is transmitted between two activities as a sequence of bytes, during the *setData*() message operation invocation. The new activity, then, writes the byte stream to the disk and un-serializes the data creating a local Plan Interpreter instance. This approach makes the Plan Interpreter easy to be stopped and resumed. The implementation of the CORBA server’s loading and saving mechanism is implemented by the LOA, using this same approach.

Network delays are not significant in our application, since most of the activity processing time will be spent in user interaction with the invoked applications.

4.3. Mobile Object Usage

The mobile object approach allows the moving of data and processing to the actor’s host. The activity manager is independent and autonomous to enact the workflow. As there is no code mobility, the workflow runtime environment, which includes the WONDER objects binaries, is replicated in each host that participates in our application execution, being restrict to the corporation using our architecture. Hence, the problems related to mobile agent's authentication and hosting do not exist.

Compared to a centralized system, the use of distributed agents carrying their own data does not reduce the overall traffic of data in the network. In both cases, data or part of the data must be copied locally, in the client hosts. The decentralized model, however, distributes the data traffic over the local network, unloading the central server backbone. The traffic is not client-server centric but peer-to-peer centric. The decentralization of data and control also distributes the server processing and communication among client hosts.

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 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.

6. Conclusions

In this paper, we have presented WONDER, a distributed architecture for large scale workflow enactment. The architecture is based on the idea that the case moves from user host to user host, following the process definition. The case is implemented as a mobile object, 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 contemplated. Such decentralization of control and data allows for the definition, enactment and management of large-scale workflows, providing the necessary scalability 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 Java language facilitates the mobile object implementation, allowing the serialization/de-serialization of the case state and data, besides being portable among different hardware and operating system environments.

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.

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.

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] "The Workflow Reference Model", Version 1.1, WFMC-TC-1003, Nov. 1994

[2] "Terminology & Glossary", Version 2.0, , WFMC-TC-1011, Jun. 1996.

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

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

[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] 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.

[7] Mohan C., Alonso G., Günthör R., Kamath M., Reinwald B. "An Overview of the Exotica Research Project on Workflow Management Systems", *Proc. 6th Int. Workshop on High Performance Transaction Systems*, Asilomar, Set. 1995.

[8] 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.
[9] "The Common Object Request Broker: Architecture and Specification" - *OMG* - Revision 2.0 July 1995.
[10] Joint submission, "OMG Workflow Management Facility", OMG dtc/99-07-05, Jul., 30, 1999.
[11] Weissenfels J., Wodtke D., Weikum G., Dittrich A. - "The Mentor Architecture for Enterprise-wide Workflow Management", University of Saarland, Department of Computer Science, 1997.
[12] Stewart R., Rai D., Dalal S. – "Building Large-Scale CORBA-Based Systems" - Component Strategies pp.34-44, 59, Jan. 1999.
[13] Weather S., Shrivastava S., Ranno F.– "CORBA Compliant Transactional Workflow System for Internet Applications" – Proc. Middleware'1998, pp. 3-17

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

Annex A

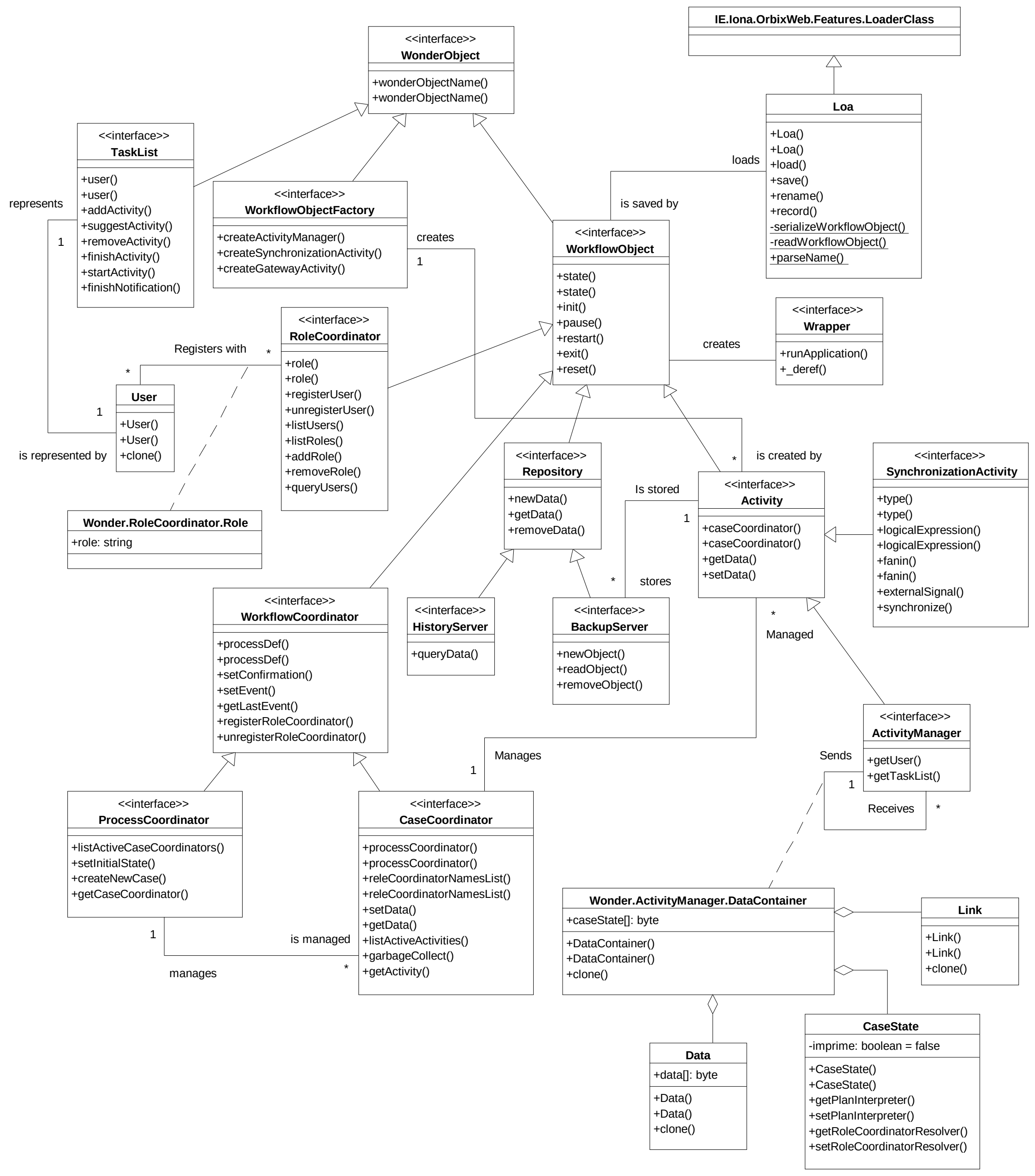

Figure A.1. The WONDER Interface (and classes) diagram