

# Workflow Execution in Disconnected Environments

Fábio Meira de Oliveira Dias, Marco Antonio Casanova,  
and Marcelo Tílio Monteiro de Carvalho

Grupo de Tecnologia em Computação Gráfica / TeCGraf,  
Pontifícia Universidade Católica do Rio de Janeiro,  
Rua Marquês de São Vicente 225, Rio de Janeiro, RJ, Brasil, CEP 22453-900  
{fmdias, casanova, tilio}@tecgraf.puc-rio.br

**Abstract.** Workflow management systems are frequently used for modeling, monitoring and controlling the coordinated execution of activities performed by workgroups in a variety of contexts. With the widespread use of portable computers and their growing computational power, conventional systems have often proved to be overly restrictive, effectively limiting the level of autonomy of the users involved. This paper identifies and analyzes different flexibilization techniques and mechanisms that can be employed in a workflow management system to better support disconnected operation. In order to test the viability of the ideas discussed, a system was built whose design met the requirements presented in the text and which allows the exploration of specific features of different kinds of workflow so as to enhance execution flexibility, without compromising the predefined structure. The system was developed in Java, which allows for portable execution and, with recent advances in compact virtual machines, enables it to run in an increasing number of devices.

## 1 Introduction

The new scenario created by the increased use of laptops and PDA's (Personal Digital Assistants) forced the reevaluation of techniques traditionally employed in the development of distributed applications. Many existing solutions do not work satisfactorily in the presence of frequent disconnections and in heterogeneous environments. Satyanarayanan [21] argues that several limitations are not due to the state of current technology, but rather they are intrinsic to any system subject to a degree of mobility.

Several applications indeed require support to mobile cooperative work with disconnection. As examples, we may quote field research operations in remote areas, where there is little or no available infrastructure, military teams engaged in intelligence or attack exercises, production or decision processes and emergency response teams.

Focusing now on the design of workflow management systems, disconnection and decentralized execution pose interesting challenges.

The first significant project regarding distributed execution of workflows was Exotica/FMQM (FlowMark on Message Queue Manager) [3]. The system was later modified to take into account the existence of disconnected clients (Exotica/FMDC) [2]. Domingos [8] proposed a hierarchical structure in which there is a core system connected to a high-speed network, which maintains the official information about the execution of workflows and data

updates executed by the participants. The CoAct (Cooperative Activity Model) transaction model [15] was also extended to support mobile users. Grigori [10] proposed combining a cooperative protocol with a traditional workflow model.

The efforts described above adopted different strategies to deal with workflow execution. In some cases, the emphasis was on the distribution of the process over several machines, without taking into account disconnection. In other cases, disconnection had to be programmed, which is not a viable alternative in mobile environments and may even permanently stop the execution, when coupled with pessimistic strategies. Also, a lack of process structuring is frequently found, mainly in solutions rooted in cooperative processes, making it harder to analyze and, in certain cases, even limiting the applicability of the mechanisms developed.

We describe in this paper the major decisions that guided the design of D2WMS, a workflow management system that works in a distributed environment, subjected to disconnections. D2WMS seamlessly integrates several strategies to address the challenges of such an environment, and here lies the major contribution of the paper. Indeed, most of the solutions found in the literature for such an environment were not fully implemented, or were adapted from solutions originally designed for less complex environments or addressed only part of the problem. By contrast, the solution proposed here has a fully operational, concise implementation.

D2WMS features a hierarchical execution model in which each instance is able to delegate the execution of subworkflows to other active instances. The execution model also allows an activity or a subworkflow to be delegated to more than one instance, simultaneously, and offers flexible consensus mechanisms. This model is explored to increase execution flexibility and to adequately deal with disconnections.

The system incorporates a service discovery mechanism and an event treatment module to enhance the proper treatment of disconnections, and adopts an optimistic replication strategy to handle workflow context variables. This infrastructure offers automatic conflict detection and resolution, seamlessly integrated with the workflow execution engine.

In addition to the above characteristics, the concise implementation of D2WMS makes it possible to use the system in different devices, making no distinction between servers or clients.

The paper is organized as follows. Section 2 describes a typical scenario, which illustrates and motivates the implementation. Section 3 covers the basic definitions needed for a complete understanding of the issues discussed. Section 4 describes in detail the major decisions that guided the design of D2WMS. Finally, Section 5 contains the conclusions.

## 2 Typical Scenario

The work described in this paper was motivated by the needs of a real project whose goal is to develop a software to support emergency response teams for an oil company, a pipeline operation company and gas distribution companies [6, 7].

The groups of people involved act according to emergency plans comprised of a structured collection of actions that must be followed to adequately respond to an anticipated set of crisis scenarios. The plans must also describe the resources required, including human and material resources, and ancillary documentation, such as maps, simulation results, lists of authorities to contact, etc. An emergency plan deployment system must therefore help monitor the execution of emergency plans and organize and access the often-vast collection of resources and documents. In this paper, we describe the execution engine developed, which consists of a workflow management system with built-in support for disconnected operation.

The emergency plans are structured in a workflow, which contains the activities that must be performed during an emergency situation. According to their nature, these activities are assigned at runtime to different teams. The required tasks typically involve the displacement of these teams in order to successfully perform the necessary actions. Therefore, the support for disconnected operation is extremely useful in such a system.

A typical scenario begins with the receiving of an alert and the identification of an emergency. Operations must then be stopped at the site and an observer is usually sent to confirm the alert. Upon confirmation, a series of steps must be followed until the end of the emergency. If we consider, for example, the case of an oil spill, contention barriers must be launched, oil pumping must be stopped, cleaning procedures must be executed, and several other combat actions must be performed. In more serious cases, fire brigades are involved, along with clinics and hospitals, and public relations personnel. The underlying workflow management system must, therefore, facilitate the coordination and collaboration between the teams involved, making it possible for the users to successfully combat the existing emergency.

In most cases, to effectively achieve their goals, the teams in charge of the operations must act distant from each other and from the infrastructure available in their offices. At the same time, they are usually expected to follow certain procedures and to record their proper execution. The role of these procedures is to help the fulfillment of the tasks or to dictate what must be done. Therefore, the application of distributed workflow execution in such a context is clearly advantageous, as it helps and controls the enactment of the required processes.

### 3 Definitions

The main goal of a workflow management system is to ensure the execution of certain activities by the right people at the correct moment. In many cases, auxiliary programs can automatically perform some activities. The Workflow Management Coalition (WfMC) is an international organization whose mission is to promote the use and to define standards for workflow management systems. According to the WfMC, a workflow management system is described as follows [25]:

A system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications.

The WfMC also established a common terminology [25]. According to it, a *process* refers to what is intended to happen, and is defined by a *process definition*, which is used by the *workflow management system* to create and manage *process instances*, which are in turn a representation of what is actually happening and include one or more *activity instances*. These can be *work items*, which are tasks allocated to a workflow participant, or *invoked applications*, which are computer tools or applications used to support an activity. Finally, process definitions can be composed of *subprocesses* in a recursive way.

A workflow process definition is used to specify the activities to be executed and the order in which this is supposed to be done. For that, conditions are usually specified in the definition, which correspond to causal dependencies between the tasks. The WfMC identifies the following four basic types of relationship between the activities:

- Sequential: several activities are executed in sequence in a single thread of execution;
- Parallel: two or more activity instances are executed in parallel within the workflow, giving rise to multiple threads of control;
- Conditional: a single thread of control makes a decision upon which branch to take when encountered with multiple alternative workflow branches;
- Iteration: a workflow activity cycle involving the repetitive execution of one (or more) workflow activity(ies) until a condition is met.

During execution, logical expressions may be evaluated by the workflow engine to decide the sequence of activity execution within a process. These expressions are evaluated against a set of *context variables*, which are specified in the workflow definition and whose values can

be altered during execution of the workflow. These expressions are commonly referred to as *transition conditions*.

The current trend of Web services flow languages [4,23] is to express the concepts described above by means of XML-based languages. The implementation described here is neutral with respect to the actual representation of a given process, therefore enabling the use of distinct workflow languages, as long as there is a correspondence between the constructs involved.

In our implementation, we followed the proposal described in [1] and modeled a workflow based on Petri net concepts [18]. Aalst argues that the use of Petri nets as a standard basis for the representation of workflows has several benefits, among which are its formal semantics, its graphical nature, its expressiveness, the availability of many analysis techniques, and a firm mathematical foundation.

To be able to model a workflow process based on Petri net concepts, a mapping is needed which establishes the equivalence between these concepts and those traditionally used in workflow systems. In this way, the semantics of such mapping can be used as the basis for the execution of a workflow. In summary, tasks are modeled by transitions and conditions are modeled by places. For a detailed discussion, the user is referred to [1].

## **4 The Design of D2WMS**

In this section, we describe in detail the major decisions that guided the design of D2WMS. The system was developed in Java, which allows for portable execution and, with recent advances in compact virtual machines, enables it to run in an increasing number of devices.

### **4.1 Workflow Definition**

As mentioned in Section 3, a workflow definition is modeled as a Petri net. It is important to notice that this representation does not limit the definition to a specific format. Actually, the workflow definition can be stored in whatever format is desired. It is only necessary to devise a module capable of translating the definition from one specific format to the one the system uses.

In the current implementation, a workflow definition is represented by a graph whose elements can be of two types: `Place` or `Transition`. A `Transition`, in turn, can be a `Workflow` or a `Task`. A `Place` works as a container for tokens, which determine the control flow during execution of the workflow. At the same time, each `Workflow` can be a subgraph

corresponding to the definition of a subworkflow. Finally, an `AutomaticTask` represents an automatic activity whose execution is performed by means of a call to a method called `execute` defined in the `Action` interface. These classes are illustrated in Figure 1.

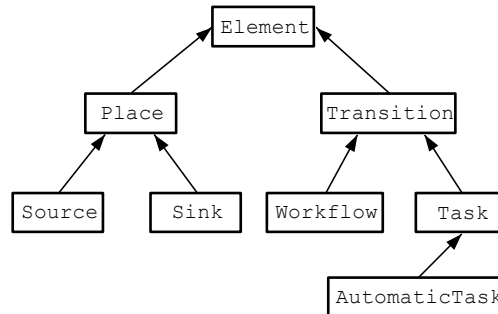


Figure 1 - Class diagram for a workflow definition.

## 4.2 Execution Model

To increase flexibility, a number of architectures were developed along the lines of *adaptable workflows* [5,10,13]. As our focus here is the execution of workflows in disconnected environments, execution flexibilization is not only interesting, but also necessary.

D2WMS supports a hierarchical execution model, in which each instance is able to delegate the execution of subworkflows to other active instances. As each subworkflow can be composed by other subworkflows, it is possible to delegate the execution of each one of them to different instances of the system, recursively. Therefore, the delegated instances have the autonomy to manage the execution of any workflow under its responsibility, be it based upon a complete definition or just part of a larger definition.

The execution model may be abstracted as a tree in which the edges represent a delegation. Intermediary nodes act simultaneously as servers and clients, with their parent nodes being the servers and their child nodes being the clients. When the execution of each delegated workflow terminates, the event is informed to the instance from which the delegation originated. Eventually, the execution is completed in the node corresponding to the root of the delegation tree.

This hierarchical execution model indeed offers increased flexibility. However, we must analyze its behavior in the presence of disconnections, specifically, what happens when: (1) the execution of a subworkflow ends and it is not possible to contact the originating instance; and (2) the disconnection periods lasts too long. D2WMS deals with the first problem by postponing the actual termination to the moment when this communication is possible and with the second problem by implementing a leasing mechanism according to which each del-

egation is subject to an expiration time. When this period is over, the subworkflow is made available once again. Therefore, in case there is any equipment failure or an excessive delay in the execution of a critical activity, the execution of the workflow as a whole does not risk being permanently hindered.

The execution model also allows an activity or a subworkflow to be delegated to more than one instance, simultaneously. The user decides when to adopt multiple delegation either at workflow design time or during workflow execution. In the current implementation, execution of the (parent) workflow proceeds when the first instance of the subworkflow successfully terminates. Different strategies might be adopted as well, such as waiting for the termination of all subworkflow instances or requiring a given quorum (using the mechanisms of Section 4.5).

The scenario described above might have delicate consequences when used inappropriately. If a subworkflow is delegated to more than one instance and each one of them further delegates other nested subworkflows, a large number of cancellations might result, for example, when the first successful termination reported automatically invalidates the other delegations. In order to avoid this kind of situation, the system does not permit the occurrence of recursive delegations, which does not solve the problem completely, but simplifies it considerably. Naturally, other strategies could have been adopted.

We now briefly outline the implementation. Each instance of the system has an `Engine` singleton, which is responsible for the management of the processes. Therefore, this object controls each process from the moment it is created until it ends and is responsible for passing the requests received to the appropriate process. Since there may be simultaneous accesses to the system and the RMI (Remote Method Invocation) technology used allows remote method calls to be executed in different threads, the object must be thread-safe. To deal with this problem, each request is put in a queue when received. Gradually, a `QueuedExecutor` removes these requests and executes them in a single thread of control. This strategy also enables the system to reject requests once the workload reaches a certain limit, which can be useful in portable machines with limited computing power.

Each request received by the `Engine` might generate a registration with the `ExpirationHandler`. This object is responsible for managing the leases associated with the assignment or the execution beginning of an activity and to the delegation of a subworkflow. The expiration time of each lease is indicated in the request and can be renewed. When a lease expires, the activity or subworkflow involved becomes available once again and can be restarted. When the nature of the activity or subworkflow does not permit this type of behavior, it is

possible to specify a lease that is infinite in practice. Such cases must be carefully studied during the definition of the workflow, as they might cause the interruption of the execution for an excessively long period or even permanently.

The `Engine` works in association with an `Executor`, to which the execution of automatic activities is delegated. D2WMS implements only one type of `Executor`, which works with a configurable pool of threads. To avoid excessive resource consumption, the pool may have a maximum size. If the execution of the activities takes too long, the automatic tasks that become ready for execution are kept in a queue until a thread is available to them. If, on the other hand, it takes too long for an activity to become ready, the threads are gradually deactivated.

### 4.3 Instance Discovery

Workflow management systems were first developed for centralized execution and only gradually begun to incorporate different distribution schemes. For example, some systems allow the user to specify the location of auxiliary programs used to implement automatic activities. Other systems permit the user to specify, in advance, a group of machines to execute different parts of the workflow. In both examples, the distribution schema is static, which is unacceptable in an environment with frequent disconnections.

To allow communication with local servers regardless of their location, D2WMS features a protocol that allows each instance of the system to rapidly identify other instances that are simultaneously operational. The protocol incorporates concepts from service discovery architectures, such as Jini [24] and SLP (Service Location Protocol) [11].

The discovery protocol is contained in the class `Server`. The first step, performed by a `Requester`, is to broadcast identification requests to which accessible instances reply informing of their existence. Each reply message contains a reference that permits the use of its interface for communication. Upon receiving a reply, the `StreamListener` collects the corresponding reference and adds it to its list of known instances. After that, the `AckSender` sends an acknowledgement with the reference of its associated instance, allowing further communication between the parties involved.

However, in the presence of disconnections, a reference to a remote instance does not guarantee that it can be successfully used. The correct behavior of the system would be to inform its known peer instances prior to disconnection, which might not happen due to failures. Therefore, each known instance is periodically *pinged* to check that it is still up and running. If an instance cannot be contacted, its reference is removed from the list of known instances.

#### 4.4 Event Generation

In many cases, the existence of an inter-process communication mechanism built into the workflow management system can be very useful. For example, it may be used to notify abnormal occurrences during the execution of a workflow. In general, Hagen [12] argues that the use of events can be beneficial to the integration of heterogeneous tools with no common API, such as conferencing systems and other groupware tools.

However, in a distributed environment with disconnections, the task of transmitting notifications becomes non trivial. It is not possible, for example, to determine the time that it takes for a notification to reach its destiny, to detect if the notification was lost or to guarantee the order in which the notifications reach the destination.

D2WMS implements an event treatment module that allows the registration of events of interest, such as starting and ending the execution of activities or workflows, updating context variables or (dis)connecting other instances. The module offers a generic registration and notification interface that allows its use with relatively heterogeneous components.

In the current implementation, the notifications are retained when immediate delivery is not possible and a configurable number of attempts is made until the message is finally discarded. In certain cases, however, it might be necessary to provide increased delivery guarantees and more sophisticated strategies. In particular, the module indeed permits registering intermediaries to relay the notification generated by the originator of the event to its final destination

The module implements the concept of *lease* [9], similar to that of Jini [24] and other similar systems, coupled with a renewal mechanism that permits to extend the period during which the registrations are valid. Together, these features enable the development of sophisticated strategies that take into account the fact that some machines might remain disconnected for a period longer than expected or even indefinitely.

In our implementation, an `EventBroker` is responsible for dealing with event generation. An `ExpirationHandler` keeps the valid registrations in memory and a `RenewalHandler` performs the periodic renewal of the registrations made by each instance of the system. Table 1 summarizes the types of events implemented.

**Table 1** - Basic kinds of events available.

Event	Description
PeerConnected	Another instance of the system found
PeerDisconnected	Contact lost with another instance of the system
PeerShutdown	An instance of the system being shut down
ProcessCreated	Workflow execution process created
ProcessFinished	Workflow execution process finished
TaskAvailable	Task available for execution
TaskAssigned	Task assigned to a participant
TaskStarted	Task execution started
TaskFinished	Task execution finished
WorkflowAvailable	(Sub)workflow available for execution
WorkflowAssigned	(Sub)workflow assigned to a participant
WorkflowFinished	(Sub)workflow execution finished
VariableChanged	Alteration of a context variable

#### 4.5 Data Replication

Each workflow has a set of context variables, which determine the control flow during execution. Therefore, to achieve the desired level of autonomy of each delegation, the system must create and maintain replicas of the context variables.

In the presence of failures or intermittent communication, optimist replication schemes offer a number of advantages over pessimist schemes. In particular, they improve user autonomy, as they allow reading and writing of any data at any time. On the other hand, they possibly lead to inconsistent local states, thereby requiring consensus mechanisms to guarantee convergence to a common state.

In view of these observations, D2WMS uses an optimist replication strategy to handle replication of workflow context variables. A replica is created whenever a delegation occurs and is destroyed when the delegation ends upon notification of the originating instance. When a delegation is about to complete, a consistency check is triggered. Conflict resolution can be either manual or automatic, using one of the simple algorithms the system implements. In both cases, it is the responsibility of the delegated instance to ensure that conflict resolution does not compromise the execution of the subworkflow involved. In some cases, it might be necessary to perform compensating steps, which must be included in the workflow definition. This type of strategy is frequently used to handle exceptions in workflow execution [14,23].

Table 2 summarizes the resolution methods available via the `ConflictResolver` developed. The algorithms are similar to those usually found in commercial database management systems replication schemes [17,19].

**Table 2** - Available conflict resolution methods.

Method	Description
ADDITIVE	Adds the difference between the original and the current values in the delegated instance to the current value in the originating instance.
AVERAGE	Averages the current values in the originating and delegated instances.
MAXIMUM	Chooses the greatest between the current values in the originating and delegated instances.
MINIMUM	Chooses the smaller between the current values in the originating and delegated instances.
EARLIEST	Chooses the earliest between the current values in the originating and delegated instances.
LATEST	Chooses the latest between the current value in the originating and delegated instances.
OVERWRITE	Discards the update to the originating instance and retains the current value in the delegated instance.
DISCARD	Discards the update to the delegated instance and retains the current value in the originating instance.
MANUAL	The user decides which value will be used.

#### 4.6 Testing the system

We developed a test environment using Python [16], a high-level scripting language with a Java implementation, that permits creating scripts that easily interact with Java objects. By exploring this environment and the modularization of the system, we generated different scripts to verify that the performance of the implementation reached satisfactory level and that the design goals were met.

By manipulating the objects available (see Section 4.1), scripts were used to emulate complete workflow definitions, with manual and automatic activities, together with subworkflows and fictitious data. Scripts were also used to test the interaction between various instances of the system and their eventual disconnection.

The test environment also enabled the use of the system via a console, making experimentation particularly easy by direct manipulation of the existing API and alleviating the need for a sophisticated graphical user interface. Figure 2 shows a simple script that creates a workflow with four activities ( $t_1$ ,  $t_2$ ,  $t_3$  and  $t_4$ ). Activities  $t_1$  and  $t_4$  are automatic and are the first and last ones of the workflow, respectively. Between them, activities  $t_2$  and  $t_3$  are manual and should be executed in parallel.

---

```

-- Creation of the workflow wf
wf = Workflow(1, 'demo wf')
-- Source and sink places
so = Source(2)
si = Sink(3)
-- First activity: automatic
t1 = AutomaticTask(4, wf.id)
-- Action that prints the description of the activity
class CustomAction(Action):
    def __init__(self, desc):
        self.description = desc
    def execute(self):
        print self.getName()
    def getName(self):
        return self.description
-- Associates the action above with activity t1
t1.setAction(CustomAction('hello, world!'))
t2 = Task(5, 'parallel task: 5')
t3 = Task(6, 'parallel task: 6')
t4 = AutomaticTask(7, wf.id)
t4.setAction(CustomAction('the end'))
-- Creates an array of places to build the workflow
p = Place.create(4, 8)
-- Creates the connections between the elements
wf.addConnector(so, t1)
wf.addConnector(t1, p[0])
. . .
wf.addConnector(p[3], t4)
wf.addConnector(t4, si)

```

---

**Figure 2 - Example of a simple script.**

## 5 Conclusions

Designing workflow management systems that support disconnected operation in a distributed environment is becoming increasingly important. Indeed, continuous access to information and a satisfactory degree of mobility are frequently considered mandatory. However, the dynamic character of this sort of environment raises a number of requirements that must be taken into account from the onset for the implementation to be successful.

The system described in this paper, D2WMS, seamlessly integrates several strategies to address the challenges of such an environment, and here lies the major contribution of the paper.

The design of D2WMS results from two major approaches. First, the system features an execution model, described in Section 4.2, based on the concepts of subworkflow and delegation. The model permits adjusting the degree of flexibility to the desired level, while supporting workflow execution in the presence of disconnections. Second, the system features a

carefully designed infrastructure, described in sections 4.3, 4.4 and 4.5, to deal with the major problems of instance discovery, event handling and data replication.

Finally, we stress that the system increases user autonomy, which is not usual in most current workflow management systems. Despite having to follow the workflow definition, the system offers the user a runtime mechanism that allows him to delegate subworkflow instances to other users or machines, with no static predefined configuration.

## Acknowledgments

This work was partially supported by CNPq under grant no. XXXXX, for Fábio Meira de Oliveira Dias, and under grant no. 552040/02-9, as part of the TerraLib Project.

## References

1. van der Aalst, W. M. P.: The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*. (1998) 21–66.
2. Alonso, G., Günthör, R., Kamath, M., Agrawal, A., Abbadi, A., Mohan, C.: Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System. *Proc. 3<sup>rd</sup> Int’l Conf. on Cooperative Information Systems*. Vienna (1995).
3. Alonso, G., Agrawal, A., Abbadi, A., Mohan, C., Günthör, R., Kamath, M.: Exotica/FMDC: A Persistent Message-Based Architecture for Distributed Workflow Management. *Proc. of the IFIP WG8.1 Working Conf. on Information Systems Development for Decentralized Organizations*. Trondheim, Norway (1995).
4. Andrews, T., Curbera, F., Dholakia, H., Golland, Y., Klein, J., Leymann, F., Liu, K., et. Al.: *Business Process Execution Language for Web Services (Version 1.1)*. BEA, IBM, Microsoft, SAP and Siebel. (2003).
5. Bussler, C.: Adaptation in Workflow Management. *Proc. of the 5<sup>th</sup> Int’l Conf. on the Software Process (ICSP5)*. Computer Supported Organizational Work. Illinois, USA. (1998).
6. Carvalho, M.T., Casanova, M.A., Torres, F., Santos, A.: INFOPAE - an Emergency Plan Deployment System. *Proc. International Pipeline Conference*. Calgary, Alberta, Canada (2002).
7. Casanova, M. A., Coelho, T. A. S., Carvalho, M. T. M., Corseuil, E. T. L., Nóbrega, H., Dias, F. M., Levy, C. H.: The Design of XPAE – An Emergency Plan Definition Language. *Anais do IV Workshop Brasileiro de Geoinformática (GEOINFO)*. Caxambu, Minas Gerais, Brasil (2002) 25–32.
8. Domingos, H. J., Martins, J. L., Preguiça, N., Duarte, S.: A Workflow Architecture to Manage Mobile Collaborative Work. *Actas do Primeiro Encontro Português de Computação Móvel*. (1999).

9. Gray, J., Cheriton, D.: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. Proc. of the 12<sup>th</sup> ACM Symp. on Operating Systems Principles. ACM Press. (1989) 202–210.
10. Grigori, D., Skaf-Molli, H., Charoy, F.: Adding Flexibility in a Cooperative Workflow Execution Engine. High Performance Computer and Networking (HPCN Europe 2000). Amsterdam, The Netherlands (2000) 227–236.
11. Guttman, E.: Service Location Protocol: Automatic Discovery of IP Network Services. IEEE Internet Computing. 3(4). (1999) 71–80.
12. Hagen, C., Alonso, G.: Beyond the Black Box: Event-based Inter-Process Communication in Process Support Systems. Proc. of the 19<sup>th</sup> IEEE Int'l Conf. on Distributed Computing Systems. Austin, Texas. (1999).
13. Han, Y., Sheth, A., Bussler, C.: A Taxonomy of Adaptive Workflow Management. CSCW Workshop on Adaptive Workflow System. Seattle, USA. (1998).
14. Jablonski, S., Bussler, C.: Workflow Management: Modeling Concepts, Architecture, and Implementation. International Thomson Computer Press. (1996).
15. Klingemann, J., Tesch, T., Wäsch, J.: Enabling Cooperation among Disconnected Mobile Users. Conf. on Cooperative Information Systems. (1997) 36–46.
16. Lutz, M.: Programming Python, 2nd Edition. O'Reilly (2001).
17. Microsoft Corporation: Microsoft SQL Server 2000 Reference Library. Microsoft Press (2000).
18. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proc. of the IEEE. 77(4). (1989) 451–580.
19. Oracle Corporation: Oracle9i Advanced Replication, Release 2 (9.2). Part No. A96567-01 (2002).
20. Pitoura, E., Samaras, G.: Data Management for Mobile Computing. Kluwer Academic Publishers (1998).
21. Satyanarayanan, M.: Fundamental Challenges in Mobile Computing. Proc. of the 15<sup>th</sup> Annual ACM Symp. on Principles of Dist. Computing. (1996) 1-7
22. W3C Note: XML Pipeline Definition Language (Version 1.0). (2002).
23. Waechter, H., Reuter, A.: The ConTract Model. In: Elmagarmid, A. (ed.): Database Transaction Models for Advanced Applications. Morgan Kaufmann Publishers, San Mateo (1992) 219–263.
24. Waldo, J.: The Jini architecture for network-centric computing. Comm. of the ACM. 42(7). ACM Press (1999) 76–82.
25. Workflow Management Coalition (WfMC): Workflow Management Coalition Terminology and Glossary. Technical Report WFMC-TC-1011 3.0. Brussels (1999).