Pragmatic Issues in Coordinated Execution and Failure Handling of Workflows in Distributed Workflow Control Architectures[†]

Mohan Kamath[‡]	*Krithi Ramamritham*
Data Server Division	Dept. of Computer Science
Oracle Corporation	University of Massachusetts
Redwood Shores, CA 94065	Amherst, MA 01003
mkamath@us.oracle.com	*krithi@cs.umass.edu*

Abstract

Recently there has been an increasing interest in workflow correctness and failure handling requirements, mostly in the context of centralized workflow control architectures. Centralized workflow control however suffers from reliability and scalability problems due to the use of a central engine to manage state information of all workflows in progress. A distributed workflow control architecture on the other hand offers much better reliability and scalability. However, developing mechanisms for handling correctness and failure requirements for distributed workflow control requires careful consideration of a number of pragmatic issues. In this paper we discuss some of these issues based on the experience gained while designing a rule-based prototype distributed workflow management system. Agents in distributed workflow control are responsible for both step execution and managing workflow state information. We first classify agents into different types and discuss the functionality to be provided by each type of agent. We then discuss how events are to be propagated between the agents as efficiently as possible while still satisfying the failure handling and coordinated execution requirements. Finally, we present a detailed discussion on performance issues. We analyze the performance of a distributed workflow control system during normal execution and when special correctness/failure handling requirements are to be satisfied. We also compare and contrast the performance of a distributed workflow control architecture with respect to central and parallel workflow control architectures.

Keywords: Workflow Management, Workflow Architectures, Coordinated Execution, Exception Handling, Recovery, Resource Utilization, Performance

† Supported by NSF grant IRI-9619588

‡ This work was done as part of the author's Ph.D.

1 Introduction

Workflow management has emerged as the technology to automate the execution of the steps (applications) that comprise a business process. Business processes often consist of applications that access shared resources like databases. Hence, dependencies arise due to the interaction between steps of concurrent workflows. Events happening in one workflow can potentially cause data inconsistencies in other concurrent workflows. Due to the complicated nature of business processes, several threads (branches) can be executing concurrently within a workflow. Failure of a step in one or more of the threads can create race conditions that can lead to data inconsistencies. To recover from the failure, compensating all the completed steps of a workflow and aborting the workflow may be very conservative and sometimes impractical. Since workflows often handle crucial information about the daily business activities of enterprises, a workflow management system (WFMS) should provide support for handling all of the above situations. A WFMS should allow a workflow designer to express coordinated execution requirements across workflows and also have execution support to handle those requirements. In addition, to handle failure of steps, a WFMS should provide an integrated approach for dealing with workflow rollback and re-execution. Such an approach will allow a workflow designer to customize workflow recovery with respect to both *correctness* and *performance* perspectives.

Modeling and execution support for failure handling and coordinated execution of concurrent workflows in a centralized workflow control environment has been discussed by the authors in [KR98]. In centralized workflow control, the agents that execute the steps of the workflows communicate with a central workflow engine that maintains workflow state information and navigates through all the workflow instances in progress and schedules the steps. To handle coordinated execution requirements, high level building blocks have been identified that express *mutual-exclusion* and complex *ordering* requirements across workflow steps, and *rollback dependency* across workflow instances. These building blocks can be used to specify both inter- and intra- workflow coordination. To handle failures within a workflow, a new scheme called *opportunistic compensation and re-execution* has been proposed which eliminates unnecessary recovery overheads when workflows are rolledback partially and re-executed. To customize the order in which the steps are compensated, in this scheme workflow designers can also specify *compensation dependent sets*. Based on these concepts, a workflow specification language called LAWS has been developed which allows the specification of failure handling and coordinated execution requirements. Requirements expressed in LAWS are converted into rules which are tuples containing an event, condition and action part. These rules are dynamically modified by the rule-based run-time systems, using the implementation level primitives *AddRule(), AddEvent() and AddPrecondition()* such that the high level requirements are satisfied.

In a distributed workflow environment, the workflow instances are scheduled and coordinated by the same agents that are responsible for executing the steps. By eliminating the central node for coordinating the workflow instances, a distributed workflow control architecture offers good reliability and scalability especially in organizations where thousands of workflow instances are in progress at any given time. To coordinate the execution of the steps of workflows, the agents have to communicate with each other. For example, after the execution of a step, an agent has to communicate the entire state information of the workflow that it is aware of to the agent responsible for executing the next step. To support failure handling and coordinated execution, an agent that executed a step in a workflow may have to communicate with agents other than those responsible for executing the preceding or the succeeding step of that workflow. Hence, mechanisms used in distributed workflow control for normal execution of workflows have to be enhanced for this purpose. More specifically, the message exchange protocols in distributed control have to be carefully designed such that the requirements are correctly met while maintaining scalability in terms of the number of messages exchanged between the agents. In our approach we first classify the agents into types based on whether they perform *coordination, execution* and *termination* of steps. Then we design protocols for communication between the different types of agents. We finally analyze the performance of a distributed workflow control architecture with mechanisms for coordinated execution and failure handling.

The rest of the paper is organized as follows. A brief overview of a centralized workflow control architecture is presented in section 2. Techniques for failure handling and coordinated execution in centralized workflow control is outlined in section 3. Section 4 provides the details of a distributed workflow control architecture. Section 5 discusses our approach for failure handling and coordinated execution in distributed control. Performance aspects of distributed control and its comparison with other workflow architectures in presented in section 6. Related work is discussed in section 7 and Section 8 summarizes the paper.

2 Centralized Workflow Control Architecture

A centralized workflow control architecture consists mainly of a workflow engine and application agents. In addition, there are other components for modeling, administration and monitoring. The workflow engine and the tools communicate with a workflow database (WFDB) to store and update workflow state data. The components are connected as shown in *Figure 1*. A workflow schema (workflow definition) is created by a workflow designer with the aid of the modeling tool. A workflow schema is essentially a directed graph with nodes representing the steps to be performed. A step is performed by typically executing a program that accesses a database. The arcs connecting the steps are of two types: data and control arcs. A data arc denotes the flow of data between steps while a control arc specifies the ordering sequence between two steps. The latter may also have a condition associated with it to indicate that the succeeding step is to be executed only if the condition evaluates to true. The program associated with a step and the data that is accessed by the step are not known to the WFMS and hence the steps themselves are 'black boxes' as far as the WFMS is concerned. Since a WFMS does not have access to step related information embedded in resource managers accessed by a program executed to perform a step, without any additional information a WFMS cannot determine if two steps from different workflows could have accessed the same resources in a conflicting manner, or when and whether a transaction invoked by a program committed or aborted.

Using a workflow schema as a template, several instances of the workflow can be created. The workflow engine is responsible for managing all the instances of all schemas, $i.e.$, it has to schedule the steps of the instances when they are eligible for execution. The workflow engine provides the agent with the information required to execute a step. The agent is responsible for executing the step and communicates back the results of the step to the engine. The engine maintains information about the workflows and steps in various tables in the WFDB for efficient access — *workflow class table* (for class definitions), *workflow instance table* (for instance specific state information) and *step table* (for step related information). These entries in the tables are related using suitable $keys$. The WFDB

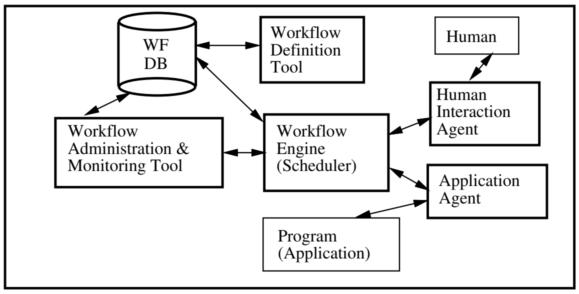


Figure 1: *Components of Centralized Workflow Control*

provides the persistence necessary to facilitate forward recovery in case of failure of the workflow engine. Additional discussion on handling WFMS component failures can be found in [AKA+94, KAGM96, KR96b]. In the rest of the paper we focus mainly on the logical failures (exceptions) in a workflow.

3 Coordinated Execution and Failure Handling in Centralized Control

In this section we briefly discuss coordinated execution and failure handling requirements and the solutions that have been developed to meet those requirements as discussed in [KR98]. One example of coordinated execution is *relative ordering* of steps from concurrent workflows. In *Figure 2(a)* steps (S12/S23 and S14/S25 respectively) from concurrent workflows WF1 and WF2 that access common resources are to be executed in the same relative order, *i.e.*, if S12 execute before S23 then S14 has to execute before S25. In this case, WF1 is the *leading* workflow and WF2 is the *lagging* workflow. Such situations arise for example in order processing workflows where orders are to be fulfilled in the sequence in which they were received. This implies that steps from two order processing workflows that conflict (*e.g.*, ordering the same parts or required the same machines) are to be executed in the same relative order. If the steps are allowed to interleave freely, a workflow processing an earlier order may not be able to continue due to lack of resources (which were used by a concurrent workflow processing a later order). Note that this is an *application* requirement since interleaving of steps will not cause data inconsistency. More detailed discussions on coordinated execution can be found in [KR98, Kam98].

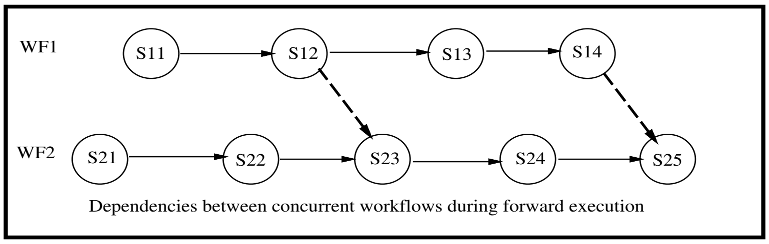


Figure 2: *Dependencies Across Workflows*

Next we consider failure handling requirements in a workflow. Consider the example shown in *Figure 3* where a step fails in a workflow with if-then-else branching. Suppose the top branch in taken after step S2 during the first execution thread and step S4 fails. The failure handling specification may require the workflow to partially rollback to step S2 and re-execute step S2. During the next execution because of changes in the data produced by step S2 (or from a feedback of failure at S4) the bottom branch may be taken. Essentially a branch that is different from the previous execution is taken. Now the effect of step S3 has to be undone. *When* S3 will be compensated depends on S3's data dependency with respect to steps S2 and S5. If there is no data dependency with either of the steps then S3 can be compensated anytime before the workflow completes, including compensating it in parallel with step S5's execution. If it has a dependency with respect to step S5 then S3 has to be compensated before the execution of step S5. The extended transaction model (Sagas) based approach requiring the compensation of all previously executed steps and aborting the workflow may be an overkill in several practical scenarios. Also since the steps of workflows are often loosely related it is not necessary to always compensate the steps in the reverse execution order. Instead a workflow designer should be able to specify such rollback requirements based on dependencies and the business logic. Hence workflow recovery customization is needed based on both *correctness* and *performance* requirements. Often in the context of partial rollback and re-execution it may not be necessary to compensate and re-execute some of the steps since they do not produce any new results or the previous results may prove to be sufficient. In such cases results from the previous execution of the steps can be *re-used* rather than compensating and re-executing the step again. This can result in substantial savings. Current workflow systems do not have provision to support such requirements. Failures in workflows can also create other types of consequences. A step failure along one thread in a workflow containing several parallel threads can cause race-conditions due to the interactions between the threads. User initiated workflow input changes and workflow aborts also form another type of failure handlign requirements. Detailed discussions on these topics can be found in [KR98, Kam98]. In summary, failure handling requirements in a workflow often require the integrated treatment of workflow rollback and forward execution.

Since we use a rule-based system to enact the workflows, to handle high level coordinated execution requirements we have identified a small set of implementation level primitives *AddRule(), AddEvent() and AddPrecondition()*, that

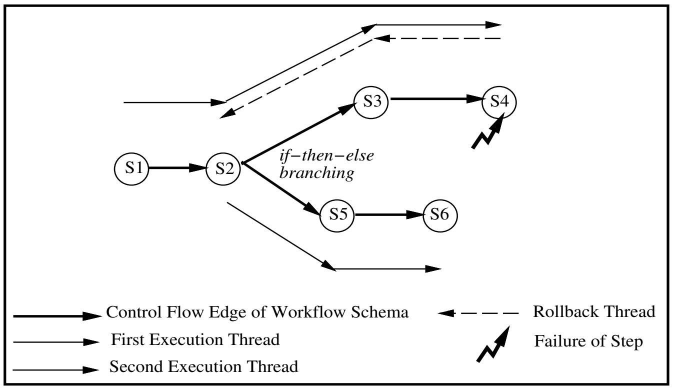


Figure 3: *Rollback in a Workflow with if-then-else branching*

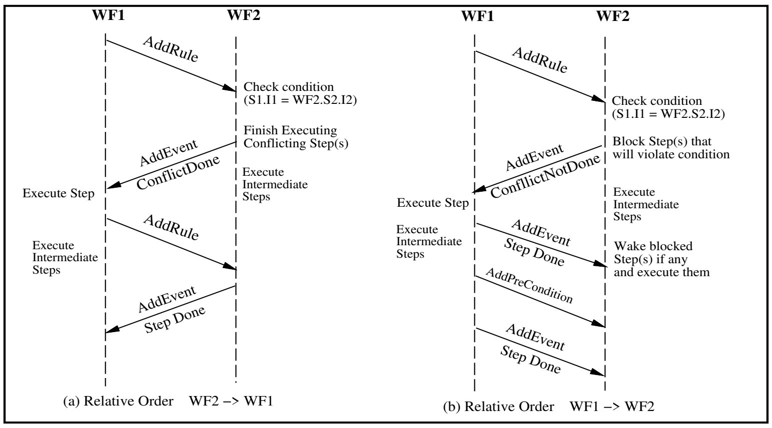


Figure 4: *Enforcing Relative Order*

affect the rule sets dynamically. These low level primitives are often used to exchange information across rule sets belonging to different workflow instances as shown in *figure 4*. The high level coordinated execution requirements are satisfied by imposing constraints on step execution. These constraints require the occurrence of additional events (often in other workflows) before a step execution can be triggered. The rules themselves are stored in the workflow database and instances of the appropriate rules are created for each workflow instance. A more detailed explanation of the above is beyond the scope of this paper and for details the reader is referred to [KR98, Kam98].

To handle failures within a workflow, we have proposed a new scheme called *opportunistic compensation and re-execution* which eliminates unnecessary compensation and re-execution of steps when workflows are rolledback partially and re-executed. Two types of compensation are possible — *complete* and *partial* and two types of re-executions are possible — *complete* and *incremental*. If the previous execution of step is useless in the current context, then a complete compensation and complete re-execution may be necessary. On the contrary, in cases where the previous execution of the step is useful an incremental re-execution or a partial compensation of the step may be sufficient to produce an effect equivalent to the complete compensation and re-execution of the step with the new inputs. To customize the order in which the steps are compensated in this scheme workflow designers can also specify *compensation dependent sets*. These specify the steps in which the compensation of one affects the others. A compensation dependent set is to be compensated only in the reverse execution order of its member steps. The notion of compensation dependent sets is different from the notion of spheres of joint compensation [Ley95]. The former is used to specify the order of step compensation in the context of our opportunistic compensation and re-execution strategy. The semantics of the latter is that when a step belonging to a sphere fails, immediately all other steps that belong to the sphere are compensated. Thus the OCR technique provides an integrated treatment of rollback and re-execution in workflows. The algorithm for the OCR technique is summarized in *figure 5*. Examples illustrating the functioning of the OCR technique can be found in [KR98, Kam98].

Next we focus on the run-time workflow control mechanisms to satisfy the specified requirement. When a workflow is instantiated using the administrative interface, a message is sent to the workflow engine. At the engine

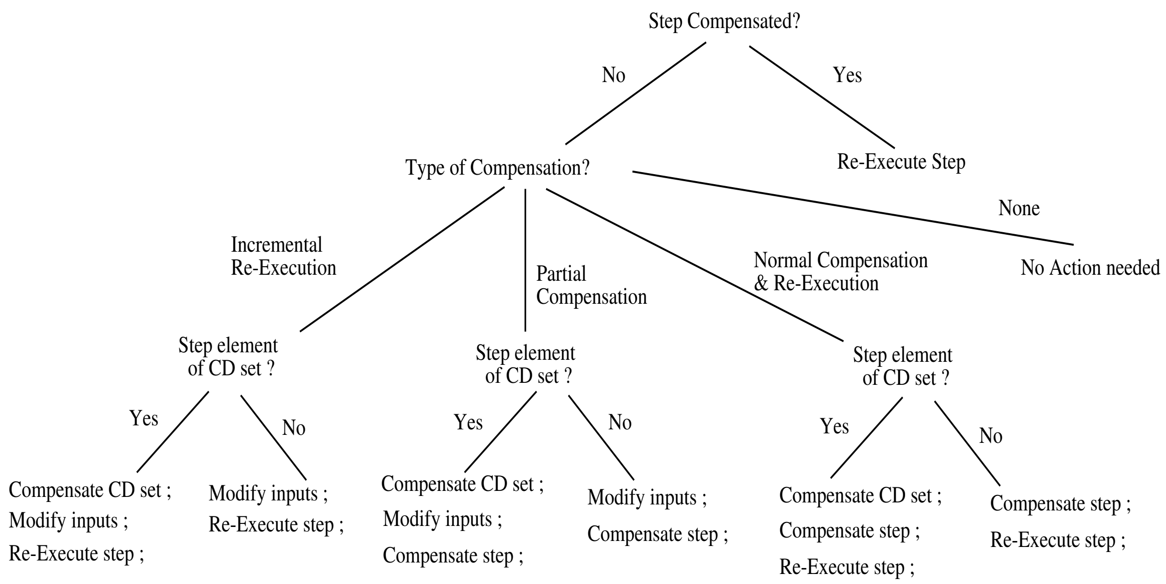


Figure 5: *Algorithm for Opportunistic Compensation and Re-execution*

a $workflow.start$ event is generated which triggers several rules that have $workflow.start$ as one of their events. Typically for the *start* steps of the workflow, this is the only event needed to trigger the $step.start$ event. Then the engine execute the steps by sending the required information (apart from the rules, step level information and data mapping information is available as part of the compiled schema) to the appropriate agent and getting back the results. If the step succeeds a $step.done$ event is generated else a $step.fail$ event is generated. This in turn triggers the appropriate rules and the execution continues until a $workflow.done$ or $workflow.abort$ event is generated. If a $workflow.abort$ event is generated then the appropriate action in taken based on the specification of the compensation dependent set and the compensation and re-execution condition. To compensate the individual steps a $step.compensate$ event is generated. Additional details can be found in [Kam98].

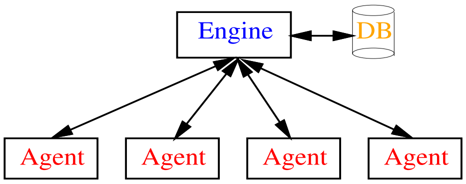
4 Distributed Workflow Control Architecture

Figure 6 illustrates the different possible workflow control architectures. A parallel workflow control architecture is an extension of the centralized architecture where several central engines work in parallel to share the load of workflow scheduling. Sine fundamentally it is not very different from the centralized architecture, we will discuss more about the parallel architecture later in the section on performance issues. In this section (and for most of the paper) we will focus only on the distributed workflow control architecture. We assume that messages are reliably delivered between agents usings tools/techniques as discussed in [AAE$^+$95]. Hence, our focus will be on the types of messages exchanged between agents and how they are processed such that the high level requirements are met.

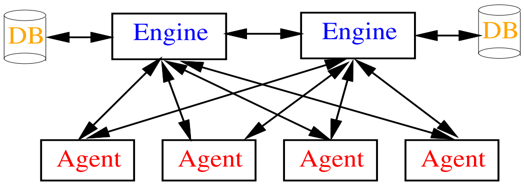
4.1 System Architecture and Agent Types

Each workflow has a *coordination agent* (not related in any way to coordinated execution as such) which is typically the agent responsible for executing the first step of the workflow. Unlike a central engine the coordination is *not* responsible for maintaining the entire state of the workflow and for performing the actual navigation of the workflow, but only for handling workflow commit and abort. The workflow coordination has the additional responsibility of maintaining the status of the workflows (whether the workflow has completed or is in execution) it started and handling user requests on those workflows. The coordination agents maintain a separate *coordination instance summary table*. The coordination instance summary table is used for handling all front end database requests including workflow status information. The front end database that provides the administrative interface to execute/abort workflows interacts only with coordination agents.

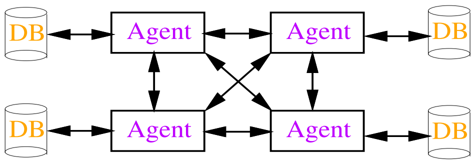
In a distributed environment, the state information of a *single* workflow is distributed across agents. Hence, tracking the state of a workflow is complicated. To determine which step of a workflow is being performed at a given instant, a chain of probe messages has to be sent starting from the agent responsible for performing the first step until the message reaches the agent that is performing the current step. Since the global state of workflow instances is also distributed across agents, it is complicated to determine whether there are conflicts/dependencies between steps of concurrent workflows managed by different workflow engines. Other than workflow packets, additional inter-agent messages are required to determine whether there are conflicts/dependencies between steps of concurrent workflows. Also, a workflow commit requires more work since the coordination agent for a workflow has to communicate with agents responsible for executing the terminal steps of each thread of execution. A *termination agent* is responsible for execution of the terminal step (last step along a path) and informing the completion to the coordination agent of the workflow to which the step belongs to. A workflow can have several terminal steps. A discussion on determining the terminal steps of a workflow is beyond the scope of this paper. We also refer to an agent responsible for executing



(a) Central



(b) Parallel



(c) Distributed

Figure 6: *Workflow Control Architectures*

a step as an *execution agent*. A coordination agent does what an execution agent does and handles workflow commit and abort. A termination agent does what an execution agent does and communicates with the coordination agent to inform the status of a terminal step. Note that each agent has an agent database (AGDB) (on the same node) in which they store all relevant persistent information such as the steps that it has executed and the corresponding results and so forth. This database also has information about agents responsible for running the steps of the various workflows and is used by the agent to determine other agents to which it has to send messages to depending on whether a step execution succeeds/fails. Also, each workflow agent can assume the roles of different types of agents concurrently since it handles steps from different workflow instances concurrently.

In a distributed workflow environment, the workflow instances are scheduled and coordinated by the agents themselves. Hence, a distributed workflow control architecture has good scalability. Since these agents not only execute the step but also handle the scheduling, they store the workflow state information in an agent database. To execute a workflow instance, agents that are responsible for executing the steps of that workflow instance have to communicate with each other as shown in *figure 6(c)*. After the execution of a step, an agent has to communicate the entire state information of the workflow that it is aware of to the agent responsible for executing the next step. This information is communicated via a *workflow packet*. The format of the messages exchanged between agents is more complex compared to that between the engine and an agent in a centralized architecture. It contains the workflow identifier, instance identifier, and the *state information* of that instance. This state information accumulates as the workflow progresses from one step to another (one agent to another). Thus the contents of a workflow packet includes the contents of the workflow packet received by the agent (request for performing that step) and the output produced by the execution of the step at the agent.

Other than the ability to process workflow packets for normal execution, agents must also be able to communicate with each other to handle failures and coordinated execution requirements. Hence the agents support a number of Workflow Interfaces (WIs). These WIs are enumerated in *Table 1*. Some of the WIs are invoked from the front end database and others from agents. All the mechanisms that will be discussed in distributed workflow control are based on these WIs — an agents exchanges messages with other agents to invoke the appropriate WIs. A summary of which of the WIs are used to handle normal execution, failure handling and coordinated execution requirements is shown in *Table 2*. The individual sections details how the WIs are used.

Since navigation is performed in a distributed fashion each node has to maintain workflow schema information and also the state information of the workflow instances whose steps it has executed. Hence, the distributed agents use the same data structures used by a central engine — workflow class tables with links to the corresponding workflow instance tables. The instance summary table in a workflow class table does not contain a status field since not all agents maintain the status information of the workflow instances, $i.e.$, whether instances are executing, completed or have been aborted.

4.2 Agent Functionality for Normal Execution

An agent listens for messages (WI calls) from other agents or the front end database. Based on the type of message received a different course of action is taken. To instantiate a workflow the front end database invokes the *WorkflowStart()* WI at the coordination agent of the workflow. When a workflow is instantiated at a coordination agent a workflow instance table is created to store the workflow state and context information. If the workflow instance is the first instance of a particular workflow class then the workflow class table is created. The workflow instance table is then linked to the workflow class table. A $workflow.start$ event is then generated which triggers rules that have $workflow.start$ as one of their events. Typically for the *start* steps of the workflow, this is the only event needed to start the step. After this, the execution of the start step and other related events are similar to those of a general step as discussed below.

Whenever a *WorkflowStart()* or *StepExecute()* WI is invoked, the data and event table of that instance is updated based on the call information. Based on the updated information in the event table, both new and pending rules are considered for firing. New rules are those that have not been fired and are considered from the general-rule table. If a new rule requires additional events before it can be fired, then it is added to the pending-rule table. Rules in the pending-rule table which have been waiting for the events present in the packet that just arrived are also fired if they do not need any other events. Rules are fired only after examining that their conditions evaluate to true. When a rule is fired it triggers the execution of a step. The information needed to execute the step such as the workflow name, instance number, step number, name of the program to be executed, and the value of input data items to the step are available locally in the workflow class and instance tables. After the step is executed, the agent then examines the results produced to check if the step was successfully executed. If the step was successful then the agent updates the data table to store the output produced by the step and generates a $step.done$ event. If the step fails, a $step.fail$ event is generated. The event table is also updated. After a step execution is complete, the state information is assembled into a packet and sent to all the agents responsible for handling the succeeding step(s) using the *StepExecute()* WI. The packet contains the data and event information present in the workflow instance table corresponding to the workflow instance the step belongs to. This contains information received by the agent from one or more agents and the information produced by the execution of the step at the agent itself. The workflow packet thus also contains event information required for the rule based navigation. We refer to the agent sending the packet as the *predecessor agent* and the agent receiving the packet as the *successor agent*. The flow of workflow packet actually corresponds to both data flow (an agent may need to send data to another agent executing a step from a different parallel branch of the same workflow) and control flow. A sample of a workflow packet is shown in *figure 7*.

Since navigation is performed by the agents themselves in a distributed manner, the various control structures $i.e.$, sequential flow, parallel branching, if-then-else branching and branch-joins as specified in the workflow schema have to be handled correctly. A rule that fires a step in a sequential path will require only the occurrence of one event (corresponding to $step.done$ of the previous step) to be fired. However, the rule may require other $step.done$ events depending on which of the steps it gets its input data from. If all the required events have not yet arrived, the rule may have to wait in the pending-rule table for future packets to arrive that carry the required events. After all the required events have occurred the rule is fired resulting in the execution of the step. In the case of an if-then-else branching,

Workflow Name: WF2
Instance Number: 4
Action: Execute S3
Data Items:
WF.I1 = 90
WF.I2 = Blower
S1.O1 = 20
S1.O2 = Gasket
S2.O1 = 45
S2.O2 = 400
Events:
WF1.S
S1.D
S2.D
R.O. Leading:
WF3.I5
WF4.I3
R.O. Lagging:
WF5.I2

Figure 7: Sample Workflow Packet in Distributed Control

after a $step.done$ event is generated on the completion of the branching step, the workflow packet is sent to the two agents that are responsible for executing the succeeding steps along the branches. After both the agents receive the packet, only one of the rules will fire based on which branching condition evaluates to true. The packets that trigger the rest of the steps in a branch operate in the same way as in a sequential path. In the case of parallel branch, the workflow packet information is sent to all the agents that are responsible for executing the succeeding steps along the various branches after a $step.done$ event is generated for the branching step. The rest of the processing at the agent is the same as that for a step on a sequential path.

Finally, consider branch-joins that occur at confluence steps. The execution of a step is triggered when control flow along all the incoming branches reaches the step. The rule that triggers the execution of a confluence step will be fired at the occurrence of $step.done$ events for the last steps of all incoming branches. Hence, the rule may have to wait in the pending rule-table for packets to arrive from all the agents that are responsible for handling the final steps of all incoming branches. As each packet arrives, the workflow instance table is updated with the data and event information contained in the message. When all the required events are available, the confluence step is executed. Loops and nested workflows are handled similarly. In the case of loops, the agent responsible for executing the last step of the loop sends a workflow packet to the first step of the workflow if the loop exit condition evaluates to false. For executing a nested workflow, the agent executing the step preceding the nested workflow sends the workflow packet to the agent responsible for executing the first step of the nested workflow. Similarly, after the completion of the nested workflow, the coordination agent of the nested workflow sends the workflow packet to the agent responsible for executing the step succeeding the nested workflow.

Another important aspect of scheduling has to do with multiple agents being eligible to execute a step as specified in the workflow schema (step table). In such a situation, a suitable successor agent has to be selected to execute the succeeding step. A simple solution would require the predecessor agent to use a two phase process to make the decision. In the first phase, it gets the state information of an agent by invoking the *StateInformation()* WI of the successor agents. The successor agents then respond with this information. The predecessor agent then picks the successor agent with for example the minimum load to execute a step. When multiple successor agents are eligible to execute a confluence step, determining the successor agent has to be handled correctly since several predecessor agents may be involved in the decision making process. Each predecessor agent obtains state information from the successor agents at different times depending on when their step completes. Hence, some of the predecessor agents may have stale information about successor agent states. Then the predecessor agents will have to exchange the state information they have about the successor agents among themselves to determine which successor agent should execute the step. A uniform and better solution is to use the "leader election" algorithm that allows the successor agents themselves determine which agent will execute the step. When all the successor agents have received messages from all the predecessor agents, they can call the *StateInformation()* WI of each other and choose the agent with the least load. This scheme also eliminates the need for successor agents to send state information to predecessor agents.

A workflow is committed when the final step along all active paths of execution have executed successfully. This implies that the final step along a sequential thread, along an if-then-else branch and along all parallel branches must be completed before a workflow is determined to be committed. Since the state information is distributed across agents, the agents handling the execution of such steps send a message to the coordination agent by calling the *StepCompleted()* WI to notify the completion of the step. The coordination agent waits for the arrival of such messages from all the agents that are responsible for executing the final steps along all active paths. In the case of nested workflows, upon the commit of a child workflow the coordination agent sends a workflow packet to the next agent responsible for executing the successor step at the parent level. Note that the step completion messages just contain steps completion information and are not the same as workflow packets.

The coordination agent knows from which of the agents it has to receive step completion messages before it determines that the workflow is committed. The coordination agent gets this information from the compilation process which can statically determine the list of such agents from the workflow schema. Once a coordination agent receives the required messages, a $workflow.done$ event is generated and the coordination agent commits the workflow. Committing a workflow corresponds to making its changes permanent. This implies that the workflow will not be rolled back in the future, $i.e.$, the effect of the individually committed steps of the workflow will not be compensated. Also, any request for aborting the workflow or changing the workflow inputs after a workflow commit will be rejected. After a workflow is committed, the instance table information is archived and the status of the instance is marked as committed in the coordination instance summary table. Periodically the coordination agents broadcast information to the other agents about the committed workflows so that they can purge their instance tables can be purged from the agents.

5 Coordinated Execution and Failure Handling for Distributed Control

From the discussion thus far, it can be observed that the lack of a single location where the entire workflow state information is available and the fact that tracking workflow state requires a chain of message between agents that execute the step of that workflow, handling special requirements such as coordinated execution and failure handling requires enhancements to the functionality of the agents and the workflow interfaces they support. In this section we discuss some of these enhancements. *Table 2* indicates the requirements for which the different workflow interfaces are used.

5.1 Enforcing Coordinated Execution

This section briefly discusses techniques for handling coordinated execution in distributed workflow control, using the *AddRule(), AddEvent() and AddPrecondition()* WIs.

Relative ordering requires significant enhancements to the mechanisms since the individual steps executed at different agents and the navigation are also performed in a distributed manner. In a central engine, the ordering information is available on the same node on which step scheduling is performed. However, in distributed workflow control, the first pair of conflicting steps is ordered at one agent and the rest of the steps to be ordered are performed at different agents. The first pair of conflicting steps is established by the agents via the *AddRule()* workflow interface using the protocol shown in *figure 4*. Based on which of the conflicting steps execute first a leading and a lagging workflow (discussed earlier in section 3) is also established. Now, successor steps of the lagging workflow have

Table 1: Types of Workflow Interfaces in Distributed Workflow Control

Supported By	*Workflow Interface*	*Invoked By*
Coordination Agent	WorkflowStart	Front End Database
Coordination Agent	WorkflowChangeInputs	Front End Database
Coordination Agent	WorkflowAbort	Front End Database
Coordination Agent	WorkflowStatus	Front End Database
Execution Agent	InputsChanged	Coordination Agent
Execution Agent	StepExecute	Coordination/Execution Agent
Execution Agent	StepCompensate	Agent
Coordination Agent	StepCompleted	Termination Agent
Execution Agent	StepStatus	Execution Agent
Execution Agent	WorkflowRollback	Execution Agent
Execution Agent	HaltThread	Execution Agent
Execution Agent	CompensateSet	Execution Agent
Execution Agent	StateInformation	Execution Agent
Execution Agent	AddRule	Execution Agent
Execution Agent	AddEvent	Execution Agent
Execution Agent	AddPrecondition	Execution Agent

Table 2: Use of WIs in Distributed Workflow Control

WI	*Used For*
WorkflowStart	Normal Execution
WorkflowChangeInputs	Failure Handling
WorkflowAbort	Failure Handling
WorkflowStatus	Normal Execution
InputsChanged	Failure Handling
StepExecute	Normal Execution
StepCompensate	Failure Handling
StepCompleted	Normal Execution
StepStatus	Failure Handling
WorkflowRollback	Failure Handling
HaltThread	Failure Handling
CompensateSet	Failure Handling
StateInformation	Normal Execution
AddRule	Coordinated Execution
AddEvent	Coordinated Execution
AddPrecondition	Coordinated Execution

to wait for the corresponding steps from the leading workflow to complete. The agents responsible for steps in the lagging workflow that have to be ordered have to be notified.

To reduce the number of messages exchanged between agents, the best way to pass ordering information to agents is to piggyback it with the workflow packet information. Thus a workflow packet contains state information and relative ordering information as well as shown in *figure 7*. The ordering information in the packets gives the instance identifiers of workflows that are leading and the workflows that are lagging. While unpacking the contents of a workflow packet, the agent checks to see if the step needs to be ordered after the execution of a conflicting step from another workflow. If so, the agent waits for a message to arrive (*AddEvent()* WI call) from the agent eligible to execute the conflicting step (this information is obtained from the workflow class table available locally). Also if there are other agents to be notified (those from lagging workflows) after the completion of the step, then the agent invokes the *AddEvent()* WI call of those agents to notify the completion. Thus information needed for coordinated execution is passed along with the other state information in the workflow packet. *AddRule()* and *AddPrecondition* calls are used similarly. For detailed discussion on the use of these calls for handling the various coordinated execution requirement, readers are referred to [Kam98].

5.2 Handling Failures

This section discusses failure handling issues which require special attention in the case of distributed workflow control. Since the steps are executed at different agents and the state information may not be available at a single node, irrespective of whether the step failed on a parallel path or a sequential path, the procedure for rolling back the workflow and re-executing the workflow is the same. This procedure automatically handles race conditions as well. Since a rollback and re-execution may affect most of the successor steps of the rollback step, all the affected threads are halted. Halting a thread corresponds to quiescing control flow along that thread. The mechanism for halting the affected threads in distributed workflow control is as follows: The agent where a step failure occurred calls the *WorkflowRollback()* WI of the agent responsible for the step to which the workflow is rolled back. This information is static and is available at the agent after the workflow schema has been compiled. None of the other agents that executed steps of that workflow are notified. As soon as an agent receives a *WorkflowRollback()* request, probe threads have to be activated to visit agents that are responsible for executing the successor steps. Hence the agent responsible for executing the rollback step calls the *HaltThread()* WI of all the successor agents of the step. The WI for an agent to request halting of a thread is HaltThread(WorkflowName, InstanceNum, OriginStep) where the $OriginStep$ is the step from which the workflow is being re-executed after a rollback. It is used in the ensuing discussion to determine which of the steps will be affected by a rollback.

When an agent receives a *HaltThread()* request it does the following. It refers to the state information of that workflow and invalidates the $step.done$ events corresponding to steps that are successors of the $OriginStep$. Also rules in the pending rule table from which the invalidated $step.done$ events have been deleted are discarded to ensure that incorrect rules will not be fired. If the agent is executing a step belonging to the instance then after the completion of the instance workflow packets are not sent to any of the successor agents. This results in quiescing the thread of control flow. If the agent has already sent workflow packets to the successor agents, then the agent in turn invokes the *HaltThread()* on the successor agents. Thus the *HaltThread()* call propagates from agent to agent until it reaches an agent that quiesces the thread. Note that as part of the rollback, events corresponding to the completion of step which are later rolledback have to be invalidated. This objective is also achieved as a side effect of halting the threads. From this state onwards, event occurrence and rule firing happen in a normal manner as the opportunistic compensation and re-execution strategy is applied. Our two pronged strategy of sending probes along the agents to halt the execution of the affected threads and invalidating old events is very simple and general enough for all types of control structures in a workflow. The invalidation strategy avoids race conditions. The WI for an agent to request compensation of a set of steps in the reverse execution order is as follows: CompensateSet(OriginStep, StepList) where the $OriginStep$ is the step from which the compensation of a dependent set is requested. It is used in the ensuing discussion.

Once the affected threads are halted each agent then uses the opportunistic compensation and re-execution strategy to re-execute the remaining steps of the workflow. Agents transfer workflow packets to successor agents using the same *StepExecute()* WI. When an agent receives a *StepExecute()* call it checks to see if the step that that is referred to in the call has already been executed. If the step has been executed then it automatically applies the opportunistic compensation and re-execution strategy, else the step is executed. The opportunistic compensation and re-execution strategy works as follows. Just as in the case of normal execution, a rule that triggers the execution of a step is fired when all the specified events events occur. However, instead of immediately executing the step, the compensation and re-execution condition is checked first to determine the exact course of action, $i.e.$, whether the step is to be partially compensated and incrementally re-executed or whether a complete compensation and re-execution is needed. The condition is evaluated by referring to the values of the different data items in the data table and step status table, including inputs to the previous execution of the step. If a re-execution is not necessary then a $step.done$ event is generated, else the step is compensated and then re-executed. If a step is compensated and it is part of a compensation dependent set then the other members of the set that executed after the step are also compensated in the reverse execution order before the step is compensated and re-executed.

The mechanism for opportunistic compensation and re-execution is very different from central workflow control and is as follows: The agent invokes the *CompensateSet(WFName,InstanceNum,OriginStep,StepList)* WI of the agent responsible for executing the last step of the compensation dependent set. On the invocation of the *CompensateSet()* WI the agent checks if the step in question has been executed. The step is also deleted from the StepList. If the step has not been executed then no action is required. If it has been executed then it is compensated using the information available locally at the agent. Then the agent invokes the *CompensateSet()* WI of the agent responsible for executing the last step of the new StepList. This procedure continues until the last remaining step (originally the first step) of the StepList is compensated. Then the workflow packet is sent to the successor agent and this procedure continues until the workflow is committed.

Another important issue during workflow re-execution has to do with if-then-else branching. If a branch different from the previous execution is taken, steps of the previously executed branch have to be compensated. Note that some of the steps on such a branch may have already been compensated since they were part of a compensation dependent set. Hence, a *compensation thread* is initiated at the branching point along the old branch by invoking the *CompensateThread()* WI . The agents successively calls this WI until all steps along the branch are compensated prior to a confluence point. The above mentioned procedure continues until the workflow is committed. If there is another step failure the same procedure is repeated.

To handle agent failures the following approach is used. Agent failures occur in two capacities — one of the predecessor agent and one of the successor agent. Let us first consider the failure of a successor agent. An agent that is responsible for executing a step preceding a given step is referred to as a predecessor agent. A successor agent is one that is responsible for executing the succeeding step. If only one agent is eligible to execute a step and it is unavailable then the predecessor waits for that agent to come up. If several agents are eligible to execute a step and if one or more of the eligible agents are unavailable then the predecessor agent chooses a successor agent from the remaining eligible successor agents. If such an agent does not exist then the agent waits to send the workflow packet until a suitable agent becomes available. The failure of a predecessor agent requires a more complex solution. For those pending-rules that require only one event and have been in that state for a set timeout period, the successor agent polls the agents responsible for executing that step about the status of the step using the *StepStatus()* WI. If the actual agent that was selected to execute the step is unavailable, the successor agent will get a response from the other eligible agents that the status as "unknown". If the step is designated as an update step then the successor agent has to wait for the failed agent to come up. Otherwise, if the step is performing a query then the successor agent requests the execution of that step by invoking the *StepExecute()* call at one of the available predecessor agents. Note that information about the type of step, $i.e.$, whether it is an update or a query is obtained from the step table in the workflow class table. A final possibility is that one of the predecessor agents respond with the status of the step as "executing" then the successor agent will wait for the packet to arrive from that predecessor agent.

The next discussion is on user initiated workflow aborts using the *WorkflowAbort()* interface on the coordination agent. A customer's cancellation order is translated into a workflow abort using the mapping information stored in the front end database. The abort request can be processed as long as the workflow has not been committed. The coordination agent verifies this fact by referring to the coordination instance summary table. Once it is determined that the abort request can be processed, steps which are to be compensated are compensated (as specified in the workflow schema). To compensate the steps the coordination agent invokes the *StepCompensate()* WI of the agents responsible for the execution of those steps. The coordination agent abort the workflow by halting the various threads of execution starting from the first step. Then the status of the instance is marked as aborted in the coordination instance summary table.

In summary, the pragmatic issues we discussed in this section with respect to coordinated execution and failure handling focus more on the protocol issues for communication between agents. Our discussion complements some of the event coordination issues discussed in [Sin96b, Sin96a] and the failure handling strategies proposed for workflow components in ORB based architectures [Wor97]. Reducing the number of messages exchanged to improve scalability is a primary concern and several techniques have been suggested in [Kam98].

6 Performance Issues

This section addresses performance related issues of the run-time mechanisms. The primary concerns include the processing load at an engine or an agent and the number of messages exchanged in the system. While the paper has primarily focused on workflow execution support in central and distributed control architectures, from a performance perspective it is necessary to consider a parallel workflow control architecture as well. A parallel control architecture is an extension of the central architecture since multiple centralized engines work in parallel to share the workflow management load. Each workflow instance however is controlled by only one workflow engine. Due to space restrictions we have not elaborated on the mechanisms required for parallel control but they are discussed in detail in [Kam98].

We first discuss performance issues related to the opportunistic compensation and re-execution strategy. To implement this strategy, the overheads include maintaining additional data that correspond to the previous execution of the steps in the data structures and checking the appropriate conditions before the execution of a step. Since these overheads are usually small, it is not expensive to use this strategy. The benefits from the strategy vary depending on the specific nature of the step. If a step involves executing simple programs that do not access many resources then the savings are not substantial. However, if the step involves either accessing a number of records from a database or human oriented tasks like moving inventory then the savings are considerable. Since many steps in real-world

Table 3: Parameters used in Analysis

Parameter	*Symbol*	*Value Range*
Number of Steps per Workflow	s	5 - 25
Number of Workflows Schemas	c	20
Number of Concurrent Instances per Schema	i	10 - 1000
Number of Engines	e	1 - 8
Number of Agents	z	10 - 100
Number of Eligible Agents per Step	a	1 - 4
Number of Conflicting Definitions per Step	d	0 - 2
Number of Steps Rolled Back on a Failure	r	1 - 10
Number of Steps to be Invalidated on a Step Failure	v	0 - 8
Number of Final Steps in a Workflow	f	1 - 4
Number of Steps to be Compensated on a workflow abort	w	0 - 4
Number of Steps/WF needing Mutual Exclusion	me	0 - 4
Number of Steps/WF needing Relative Ordering	ro	0 - 4
Number of Steps/WF having Rollback Dependency	rd	0 - 2
Navigation and Other Load per Step (# of instructions)	l	—
Probability of Logical Step Failure	pf	0.0 - 0.2
Probability of Workflow Input Change	pi	0.0 - 0.05
Probability of Workflow Abort	pa	0.0 - 0.05
Probability of step re-execution	pr	0.0 - 0.5

workflows correspond to the second variety, in general the benefits from the OCR scheme is considerable while paying a small overhead.

The rest of the analysis focuses on the load at a node and the number of messages exchanged based on various parameters and the values we have assumed for them as shown in *Table 3*. The value ranges were chosen based on intuition since performance studies related to workflow execution in the presence of failures and under different architectures are not available. *Tables 4 and 6* summarize the approximate expressions that can be used to determine the load at a node or the number of messages exchanged for the various mechanisms under central and distributed control architectures. Similar expressions have been derived for parallel workflow control architectures but not been presented here for space considerations. Interested readers are referred to [Kam98]. Using the approximate values and expressions it is possible to get a feel for how the various mechanisms compare under different architectures. In

Table 4: Load and Physical Messages in Centralized Workflow Control

Mechanism	*Expression*	*Normalized Value*
Load at Engine	**per instance**	
Normal Execution	$l \cdot s$	$15 \cdot l$
Workflow Input Change	$l \cdot r \cdot pi$	$0.125 \cdot l$
Workflow Abort	$l \cdot w \cdot pa$	$0.0500 \cdot l$
Failure Handling	$l \cdot r \cdot pf$	$0.5 \cdot l$
Coordinated Execution	$l \cdot (me + ro + rd) \cdot s$	$75 \cdot l$
Physical Messages Exchanged	**per instance**	
Normal Execution	$2 \cdot s \cdot a$	60
Workflow Input Change	$2 \cdot r \cdot pi \cdot pr \cdot a$	0.125
Workflow Abort	$2 \cdot w \cdot pa \cdot a$	0.2
Failure Handling	$2 \cdot r \cdot pf \cdot pr \cdot a$	0.5
Coordinated Execution	0	0

this analysis normal execution is treated as the situation where coordinated execution is not required. The normalized values assigned for the expressions are based on the average values of the parameters specified in *Table 3*.

The expressions for load comparison are based on the estimated number of steps or other actions that would be performed at the engine in case of central and parallel control and at an agent in the case of distributed control. Similarly, the expressions that give the number of messages physically exchanged are based on the interactions between the various nodes of the system under different architecture, $i.e.$, engine with agent in a centralized architecture, engines with engines and engines with agents in a parallel architecture and agents with agents in a distributed architecture.

- *Normal Execution:* The scheduling load is based on the number of steps to be executed for all instances of all workflows schemas. The scheduling load is shared by the engines in the case of parallel control and the agents in distributed control. Since the number of agents (z) typically far outnumber the number of engines (e), the load is moderate at a parallel engine and the lowest at a distributed agent. The number of messages exchanged in the system is the same for central and parallel control but is lesser for distributed control since the number of final steps is always far less than the total number of steps in a workflow. Thus both from the perspective of engine load and number of messages, a distributed control architecture is very scalable for normal execution.
- *Workflow Input Changes:* The load factor here depends on the probability of workflow input change and the number of steps to be rolled back. Since the load is again shared by the engines in the case of parallel control and agents in the case of distributed control, the distributed agents are the most lightly loaded. The number

Table 5: Load and Physical Messages in Parallel Workflow Control

Mechanism	*Expression*	*Normalized Value*
Load at Engine	**per instance**	
Normal Execution	$l \cdot s$ / e	3.75 $\cdot l$
Workflow Input Change	($l \cdot r \cdot pi$)/e	0.0313 $\cdot l$
Workflow Abort	($l \cdot w \cdot pa$)/e	0.0125 $\cdot l$
Failure Handling	($l \cdot r \cdot pf$)/e	0.125 $\cdot l$
Coordinated Execution	($l \cdot (me + ro + rd) \cdot s$)	75 $\cdot l$
Physical Messages Exchanged	**per instance**	
Normal Execution	$2 \cdot s \cdot a$	60
Workflow Input Change	$2 \cdot r \cdot pi \cdot pr \cdot a$	0.125
Workflow Abort	$2 \cdot w \cdot pa \cdot a$	0.2
Failure Handling	$2 \cdot r \cdot pf \cdot pr \cdot a$	0.5
Coordinated Execution	$(me + ro + rd) \cdot e \cdot s$	300

of messages exchanged is again the same for central and parallel workflow control. However, for distributed workflow control, the number of messages is very much dependent on the number of steps to be invalidated which corresponds to the number of agents to be traversed for halting the threads affected by the change. If the number of such agents to be traversed is smaller than the number of steps crossed during rollback then distributed control requires fewer messages. Thus on an average the three architectures are comparable in terms of the number of messages. Since the overall probability of workflow input changes is small, they do not affect system scalability in a big way.

- *Workflow Abort:* The mechanisms considered here are for user initiated workflow abort where some of the executed steps are compensated and the workflow is aborted. From the perspective of the load, distributed agents again have the lowest load. In terms of the number of messages exchanged, the engines know the node at which a step was executed whereas the coordination agent may have to send messages to all eligible agents (for executing the steps) to be compensated. Hence central and parallel workflow control may have an edge over distributed control depending on the number of agents eligible for executing the steps. Since the overall probability of workflow aborts is small, they do not affect system scalability in a big way.
- *Failure Handling:* The load actually depends on the probability of step failures and the number of steps crossed during workflow rollback. Since the load is shared, distributed agents again have the lowest load. In terms of the number of messages exchanged the situation is similar to that of workflow input changes and there is no

Table 6: Load and Physical Messages in Distributed Workflow Control

Mechanism	*Expression*	*Normalized Value*
Load at Engine	**per instance**	
Normal Execution	$l \cdot s$ / z	0.3 $\cdot l$
Workflow Input Change	($l \cdot r \cdot pi$)/ z	0.0025 $\cdot l$
Workflow Abort	($l \cdot w \cdot pa$)/ z	0.001 $\cdot l$
Failure Handling	($l \cdot r \cdot pf$)/z	0.01 $\cdot l$
Coordinated Execution	($l \cdot (me + ro + rd) \cdot a \cdot d \cdot s$)/z	1.5 $\cdot l$
Physical Messages Exchanged	**per instance**	
Normal Execution	$s \cdot a$ + f	32
Workflow Input Change	$(r + v) \cdot pi \cdot a$	0.45
Workflow Abort	2 $\cdot w \cdot pa \cdot a$	0.2
Failure Handling	$(r + v) \cdot pf \cdot a$	1.8
Coordinated Execution	$(me + ro + rd) \cdot a \cdot d \cdot s$	150

clear winner. If the number of agents to be traversed for invalidating previous step executions and halting the affected threads is smaller than the number of steps crossed during rollback, then distributed control requires less number of messages. Thus on an average the three architectures are comparable in terms of the number of messages.

- *Coordinated Execution:* The load primarily depends on the number of steps for which coordinated execution requirements of mutual exclusion, relative ordering and rollback dependency have been specified. Note that the loads are comparable at a central and a parallel engine since the number of engines, e, cancel out both at the numerator and the denominator. Also, the factor $a \cdot d/z$ is usually a small fraction and hence distributed agents are the least loaded. In terms of the number of messages, a central engine is clearly a winner since no physical messages are exchanged. Between a parallel engine and a distributed agent there is no clear winner. If the factor $a \cdot d$ is less than e, then distributed agents use fewer messages else a parallel engine uses lesser number of messages. Since the probability of the existence of coordinated execution requirements for a step is the factor $(me + ro + rd)/s$, coordinated execution requirements affect the scalability of parallel and distributed control architectures.

We have performed a comparison of the various mechanisms for different architectures in terms of the scheduling load and the number of messages exchanged. From the perspective of load on a node, a distributed agent is the least loaded and hence very scalable. From the perspective of the number of messages exchanged, the choice depends largely on the extent of coordinated execution requirements. The common case is that the number of steps that require

Table 7: Recommended Choice of Architectures for Various Requirements

Criteria	*Normal*	*Normal + Failures*	*Normal + Coordinated*
Load at Engine	(1) Distributed	(1) Distributed	(1) Distributed
	(2) Parallel	(2) Parallel	(2) Parallel
	(3) Central	(3) Central	(3) Central
Physical Messages	(1) Distributed	(1) Distributed	(1) Central
	(2) Parallel	(2) Parallel	(2) Distributed
	(2) Central	(2) Central	(3) Parallel

coordinated execution is small. In such a scenario, distributed workflow control is a clear choice. In the unlikely case that several steps have coordinated execution requirements then central or parallel control is preferable. Even in such cases, the performance of parallel and distributed control can be improved such that they are still scalable by enforcing certain requirements ($e.g.$, conflicting steps to be executed on the same agent). A summary of the recommended choice based on various requirements is shown in *Table 7.* The numbers indicate the preferred order of choice.

7 Related Work

The need for non centralized control of workflows has been recognized earlier in systems like INCAS [BMR96], Exotica/FMQM [AAE$^+$95], and WIDE [CGS97]. In both these pieces of work the primary focus is on supporting distributed scheduling of workflows. In addition to scheduling support for normal execution, our work discusses support for handling logical failures and coordinated execution. In the case of distributed workflow control since the state of an individual workflow is distributed across nodes, exchange of state and event information is needed not only for coordinated execution but also for handling failures of steps within a workflow. The different message exchange protocols (WIs) and techniques used in our approach ensure that the data and state information exchange is done correctly and as efficiently as possible. Thus the work presented here significantly extends previous research efforts in building correct and scalable non-centralized WFMSs.

Recent work on distributed constraints and scheduling of workflow computations [Sin96b, Sin96a] is closely related to the objective of CREW. Starting with a workflow specification as a language based on temporal logic it shows how to derive pre-conditions for events. This work is driven by pragmatic considerations and has taken the form of developing specification-level building block, implementation-level primitives, translation from the former to the latter as well as a realization of this necessary infrastructure. Our strategies also complement the failure handling strategies proposed for workflow components in ORB based architectures [Wor97]. A detailed comparison of our work with other area including coordinated execution, failure handling, rule-based system, software processes can be found in [Kam98]. In summary, our work discussed new data exchange protocols and interfaces between agents, functionality of the agents and performance of workflow control architectures, thereby complementing other research in the area of distributed workflow management.

8 Conclusions

Workflow management has emerged as the technology to automate the execution of the steps (applications) that comprise a business process. Since business processes often consist of applications that access shared resources like databases, dependencies arise due to the interaction between steps of concurrent workflows. Events happening in one workflow can potentially cause data inconsistencies in other concurrent workflows. A variety of coordinated execution and failure handling requirements arise both from the correctness and performance perspective.

This paper first highlighted the coordinated execution and failure handling requirements and presented solutions for a centralized workflow control architecture. Then it discussed the details of a distributed control architecture, where the workflow instances are scheduled and coordinated by the same agents that are responsible for executing the steps. The lack of a single location where the entire workflow state information is available and the fact that tracking workflow state requires a chain of message between agents that execute the step of that workflow, handling special requirements such as coordinated execution and failure handling requires enhancements to the functionality of the agents and the workflow interfaces they support. To support failure handling and coordinated execution, an agent that executed a step in a workflow may have to communicate with another agent that is not responsible for executing the preceding or the succeeding step of that workflow. More specifically, the message exchange protocols in distributed control have to be carefully designed such that the requirements are correctly met while maintaining scalability in terms of the number of messages exchanged between the agents. In our approach we first classified the agents based on whether they perform *coordination, execution* and *termination* of steps. Then we designed protocols for communication between the different types of agents. We finally analyzed the performance of centralized and distributed workflow control architectures from the perspective of normal execution, coordinated execution and failure handling and made recommendations about the suitability of these architectures under various scenarios. By considering several pragmatic issues such as (i) data exchange protocols and interfaces between agents, (ii) functionality of the agents and (iii) performance of workflow control architectures, the discussions in this paper complement other research in the area of distributed workflow management.

References

[AAA+96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Guenthoer, and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 1996.

[AAE+95] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör, and M. Kamath. Exotica/FMQM:A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proc. of IFIP WG 8.1 Workgroup Conference on Information Systems Development for Decentralized Organizations (ISDO95), Trondheim, Norway*, August 1995.

[AKA+94] G. Alonso, M Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Failure Handling in Large Scale Workflow Management Systems. Technical Report RJ 9913(87293), IBM Almaden Research Center, November 1994.

[BMR96] D. Barbara, S. Mehrota, and M. Rusinkiewicz. INCAS: Managing Dynamic Workflows in Distributed Environments. *Journal of Database Management, IDEA Group Publishing.*, 7(1):5–15, December 1996.

[CGS97] S. Ceri, P. Grefen, and G. Sanchez. WIDE: A Distributed Architecture for Workflow Management. In *Proc. of 7th Intl. Workshop on Research Issues in Data Engineering*, Birmingham, U.K., April 1997.

[CR90] P. K. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proc. ACM SIGMOD Conf.*, page 194, Atlantic City, NJ, May 1990.

[Dav78] C. T. Davies. Data Processing Spheres of Control. *IBM Systems Journal*, 17(2):179–198, 1978.

[EL95] J. Eder and W. Liebhart. The Workflow Activity Model WAMO. In *Proceedings of 3rd Intl Conference on Cooperative Information Systems*, Vienna, Austria, September 1995.

[Elm92] A. Elmagarmid, editor. *Transaction Models for Advanced Database Applications*. Morgan-Kaufmann, 1992.

[Geo95] Georgakopolous D. and Hornick M. and Sheth A. An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–152, April 1995.

[GHKM94] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 462, Houston, TX, February 1994.

[Hol94] D. Hollingsworth. Workflow Management Reference Model, 1994. The Workflow Management Coalition, Accessible via: http://www.aiai.ed.ac.uk/WfMC/.

[Hsu93] M. Hsu. Special Issue on Workflow and Extended Transaction Systems. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 16(2), June 1993.

[Hsu95] M. Hsu. Special Issue on Workflow Systems. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 18(1), March 1995.

[KAGM96] M. Kamath, G. Alonso, R. Günthör, and C. Mohan. Providing High Availability in Workflow Management Systems. In *Proceedings of the Fifth International Conference on Extending Database Technology (EDBT-96)*, Avignon, France, March 1996.

[Kam98] M. Kamath. *Improving Correctness and Failure Handling in Workflow Management Systems*. PhD thesis, Department of Computer Science, Univeristy of Massachusetts, Amherst, MA, may 1998.

[KR96a] M. Kamath and K. Ramamrithan. Bridging the Gap between Transaction Management and Workflow Management. In *Proc. of NSF Workshop on Workflows and Process Automation in Information Systems*, Athens, Georgia, May 1996.

[KR96b] M. Kamath and K. Ramamrithan. Correctness Issues in Workflow Managemen. *Distributed Systems Engineering Journal — Special Issue on Workflow Systems*, 3(4):213–221, December 1996.

[KR98] M. Kamath and K. Ramamritham. Failure Handling and Coordinated Execution of Concurrent Workflows. In *Proc. of Intl. Conference on Data Engineering (ICDE), Orlando, Florida*, 1998.

[KRGL97] M. Kamath, K. Ramamrithan, N. Gehani, and D. Lieuwen. WorldFlow : A System For Building Global Transactional Workflows. In *Proc. of 7th Intl. Workshop on HPTS (High Performance Transaction Systems) 97*, Asilomar, California, September 1997.

[KS94] N. Krishnakumar and A. Sheth. Specifying Multi-system Workflow Applications in METEOR. Technical Report TM-24198, Bellcore, May 1994.

[Ley95] F. Leymann. Supporting Business Transactions via Partial Backward Recovery in Workflow Management. In *Proc. of BTW'95*, Dresden, Germany, March 1995. Springer Verlag.

[LW98] D. Lomet and G. Weikum. Efficient and Transparent Application Recovery in Client-Server Information Systems. In *Proc. of SIGMOD Intl. Conference on Management of Data, Seattle*, 1998.

[M+98] P. Muth et al. From Centralized Workflow Specification to Distributed Workflow Execution. In *Journal of Intelligent Information Systems*, February 1998.

[MHG+92] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie. Distributed Object Management. In *Int. J.of Intelligent and Cooperative Information Systems 1, 1*, March 1992.

[Mos81] J. E. B. Moss. Nested Transactions: An Approach To Reliable Distributed Computing. Technical report, PhD Thesis, MIT, Cambridge, MA, April 1981.

[MSKW96] J.A. Miller, A.P. Sheth, K.J. Kochut, and X. Wang. CORBA-Based Run Time Architectures for Workflow Management Systems. *(to appear in) Journal of Database Management, Special Issue on Multidatabase Systems*, 7(z), 1996.

[RC96] K. Ramamritham and P. K. Chrysanthis. Advances in Concurrency Control and Transaction Processing, 1996.

[RS94] M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond. W. Kim, Ed.*, Addison Wesley, 1994.

[S+96] A. Sheth et al. Supporting State-Wide immunization Tracking using Multi-Paradigm Workflow Technology. In *Proceedings of the 22nd VLDB Conference*, Bombay, India, 1996.

[Sch93] F. Schwenkreis. APRICOTS - A Prototype Implementation of a ConTract System - Management of Control Flow and the Communication System. In *Proc. of the 12th Symposium on Reliable Distributed Systems, IEEE Computer Society Press*, Princeton(NJ), 1993.

[She96] A. Sheth, editor. *Proceedings of NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, `http://lsdis.cs.uga.edu/activities/NSF-workflow/`, May 1996.

[Sin96a] M. P. Singh. Distributed Scheduling of Workflow Computations. Technical Report TR-96-18, Dept of Computer Science, North Carolina State University, 1996.

[Sin96b] M. P. Singh. Synthesizing Distributed Constrained Events from Transactional Workflow Specifications. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 1996.

[SJKB94] H. Schuster, S. Jablonski, T. Kirsche, and C. Bussler. A Client/Server Architecture for Distributed Workflow Management Systems. In *Proc. of Third Int'l. Conf. on Parallel and Distributed Information Systems*, Austin, Texas, September 1994.

[SR93] A. Sheth and M. Rusinkiewicz. On Transactional Workflows. *Bulletin of the Technical Committee on Data Engineering*, 16(2), June 1993. IEEE Computer Society.

[Wor97] D. Worah. Error Handling and Recovery For The ORBWork Workflow Enactment Service in METEOR. Master's project, University of Georgia, Computer Science Department, 1997.

[WR92] H. Waechter and A. Reuter. The ConTract Model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–263. Morgan Kaufmann Publishers, San Mateo, 1992.