A Transactional Workflow based Distributed Application Composition and Execution Environment

Santosh K. Shrivastava and Stuart M. Wheater
Department of Computing Science, Newcastle University,
Newcastle upon Tyne, NE1 7RU, England.

Introduction

We have designed a fault-tolerant application composition and execution environment for distributed applications whose executions could span arbitrarily large durations. A preliminary version of the system has been implemented (both in C++ and Java) using distributed object and CORBA middleware technologies. In effect, we have implemented a *transactional workflow system* that enables sets of inter-related tasks to be carried out and supervised in a dependable manner. Our system has been designed to meet the requirements of *interoperability*, *scalability*, *flexible task composition*, *dependability*, *dynamic reconfiguration* and *auditability.*

We are targeting applications in the Internet/Web environment. The Internet frequently suffers from failures which can affect both the performance and consistency of applications run over it. The types of applications we have in mind are to do with the automation of so called 'business processes' of organizations that are increasingly using the Internet for their day to day functioning. The domain of electronic commerce is a good example: here automation of business processes would include system support to help consumer-to-business as well as business-to-business interactions for buying and selling of goods. First we describe why meeting the requirements stated above are deemed essential for such applications.

The requirements of *flexible application composition, interoperability and scalability* are derived from the observation that most such applications are rarely built from scratch; rather they are constructed by composing them out of existing applications and protocols. It should therefore be possible to compose an application out of component applications (that can be arbitrarily distributed) in a uniform manner, irrespective of the languages in which the component applications have been written, and the operating systems of the host platforms. The requirement of *dependability* is derived from the observation that the resulting applications can be very complex in structure, containing many temporal and data-flow dependencies between their constituent applications. However, constituent applications must be scheduled to run respecting these dependencies, despite the possibility of intervening processor and network failures. Furthermore, application level fault-tolerance is necessary to maintain application specific consistency of the application in the face of failure exceptions from the underlying system (e.g., inability to start an application due to some faulty condition that is refusing to go away, say a network partition that is not healing), and application level exceptions that require error recovery in the form of aborts or compensations. Next we come to the requirement for *dynamic reconfiguration*. The execution of an application may take a long time to complete, and may contain long periods of inactivity (minutes, hours, days, weeks etc.), often due to the constituent applications requiring user interactions. A long running application is likely, at some point during its execution, to encounter changes to the environment within which it is executing; these environmental changes could include machine and network related failures, services being moved or withdrawn, or the application's functional requirements being changed. Dynamic reconfiguration mechanisms that will allow applications to change their internal structures to ensure forward progress are therefore required. Finally, facilities are required for examining an application's execution history (e.g., to be able to settle disputes). So, a durable 'audit trail' recording the application's state and certain significant events that have occurred during its execution needs to be maintained.

Workflow Management Systems

Workflow systems are intended to direct, coordinate and monitor execution of tasks arranged to form *workflow applications* representing business processes. *Tasks* (*activities*) are application specific units of work. A *Workflow schema* (*workflow script*) is used explicitly to represent the structure of an application in terms of tasks and temporal dependencies between tasks. An application is executed by *instantiating* the corresponding workflow schema.

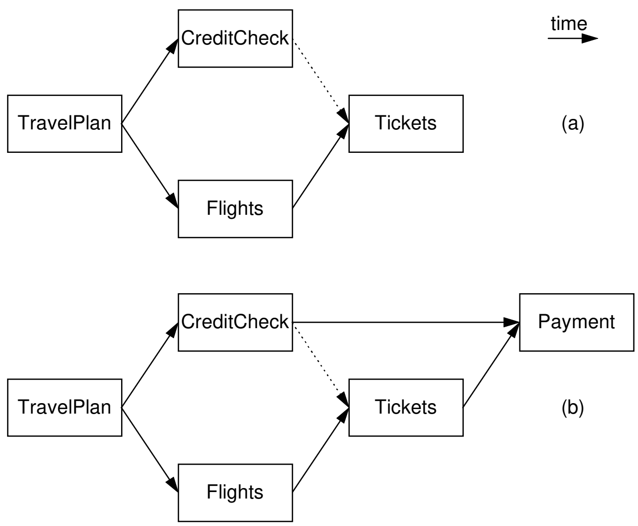

Fig. 1: Inter-task dependencies

Imagine an electronic travel booking application. Fig. 1(a) shows the business process (an 'activity diagram') of the travel agency depicting the *temporal dependencies* between its four constituent applications (or tasks), *TravelPlan*, *CreditCheck*, *Flights* and *Tickets*. Tasks *CreditCheck* and *Flights* execute concurrently, but can only be started after the *TravelPlan* task has terminated and supplied the necessary data, so these two tasks have *dataflow dependencies* on the *TravelPlan* task. Task *Tickets* can only be started after *Flights* task has terminated and supplied the necessary data and task *CreditCheck* has terminated in an 'ok' state. In this case, task *Tickets* has a dataflow dependency on *Flights*, and a restricted form of dataflow dependency (called *notification dependency*) on *CreditCheck*.

There are several organizations involved in the above application (customer organization, travel agency, credit card agency, etc.) and the application is widely distributed (e.g., *TravelPlan* is executed by the local branch of the travel agency, *CreditCheck* is executed at some credit card agency and so forth); furthermore, each constituent application itself can be a workflow. Each organization may well possess its own workflow system for carrying out its activities. A specific way of executing this application could be: the travel agency has the application description (workflow script) and is responsible for coordinating the overall execution and it itself executes tasks *TravelPlan* and *Tickets*; its workflow system will invoke *CreditCheck* task and *Flights* task at other organizations. Clearly, there is a requirement that the constituent applications must be scheduled to run respecting the dataflow and notification dependencies, despite the possibility of intervening processor and network failures: this should be met by the workflow systems.

Dynamic reconfiguration mechanisms should make it possible to change the structure of a *running application* by adding/deleting: tasks, notifications and dataflow dependencies in a consistent manner. By consistent we mean respecting two conditions: (i) modifications are carried out atomically (either all the changes are performed or none) with respect to the normal processing activities of the application; and (ii) the application is able to execute respecting these changes. Referring to fig. 1 (a), assume that the task *Flights* needs to be replaced by a task *TrainJourney*, with the same input-output dependencies; such a change can be performed provided the *Flights* task has not yet started. On the other hand, if the task *Flights* has started, then it should be possible to insert a *compensating* task *CancelFlight* followed by *TrainJourney*. Take another example: assume that electronic payment facilities are available and, it is desirable to extend the application with a new task (*Payment*) with the dependencies as shown in fig. 1(b). Once these changes have been performed, the run time system should ensure that *Payment* task does receive its inputs. So for example, if the changes are performed after *CreditCheck* has terminated, its outputs should still be made available for consumption by *Payment*.

Clearly, there is a need for a standard way of representing application structure and sending and receiving 'work items', if organizations are to cooperate. Standardization efforts are therefore underway. Currently, the Object Management Group (OMG), the consortium of IT vendors and users, is evaluating proposals for a workflow facility standard. Our system architecture is the basis of Nortel's submission to the OMG for the development of the standard [1].

System Architecture

Our system architecture is decentralized and open: it has been designed and implemented as a set of CORBA services to run on top of a given ORB. Details of our system appear in [1,2,3].

We discuss how our system has been designed to meet the requirements stated earlier, namely: interoperability, scalability, flexible task composition, dependability, dynamic reconfiguration and auditability.

- *Interoperability*: As stated before, the system has been structured as a set of CORBA services to run on top of a CORBA-compliant ORB thereby supporting interoperability including the incorporation of existing applications and services.
- *Scalability*: There is no reliance on any centralized service that could limit the scalability of workflow applications.
- *Flexible Task Composition*: The system provides a uniform way of composing a complex task out of transactional and non-transactional tasks. This is possible because the system supports a simple yet powerful *task model* permitting a task to perform application specific input selection (e.g., obtain a given input from one of several sources) and terminate in one of several outcomes, producing distinct outputs.

- *Dependability*: The system has been structured to provide dependability at *application level* and *system level*. Support for application level dependability has been provided through flexible task composition mentioned above that enables an application builder to incorporate alternative tasks, compensating tasks, replacement tasks etc., within an application to deal with a variety of exceptional situations. The system provides support for system level dependability by recording inter-task dependencies in transactional shared objects and by using transactions to implement the delivery of task outputs such that destination tasks receive their inputs despite finite number of intervening machine crashes and temporary network related failures.
- *Dynamic Reconfiguration*: Task model referred to earlier is expressive enough to represent temporal (dataflow and notification) dependencies between constituent tasks. Our application execution environment is *reflective*, as it maintains this structure and makes it available through transactional operations for performing changes to it (such as addition and removal of tasks as well as addition and removal of dependencies between tasks). Thus the system directly provides support for dynamic modification of workflows (*ad hoc workflows*) [4]. The use of transactions ensures that changes to schemas and instances are carried out atomically with respect to normal processing.
- *Auditability*: As mentioned above the workflow application's structure and state is maintained in transactional shared objects so their committed states contain a durable record of the application's execution history. These objects have CORBA interfaces that provide operations that allow examinations of the application's state, structure and certain significant events that have occurred during its execution.

The workflow management system's structure is shown in fig. 2. Here the big box represents the structure of the entire distributed workflow system (and not the software layers of a single node); the small box represents any node with a Java capable browser. The most important components of the system are the two transactional services, the workflow *repository* service and the workflow *execution* service. These two facilities make use CORBA Object Transaction Service (OTS). The implementation for OTS used for the workflow management facility is OTSArjuna, which is an OTS compliant version of Arjuna distributed transaction system built by us [5]. In our system, application control and management tools required for functions such as instantiating workflow applications, monitoring and dynamic reconfiguration etc., (collectively referred to as *administrative* applications) themselves can be implemented as workflow applications. Thus the administrative applications can be made fault-tolerant without any extra effort. Graphical user interface to the these administrative applications has been provided by making use of Java applets which can be loaded and run by any Java capable Web browser.

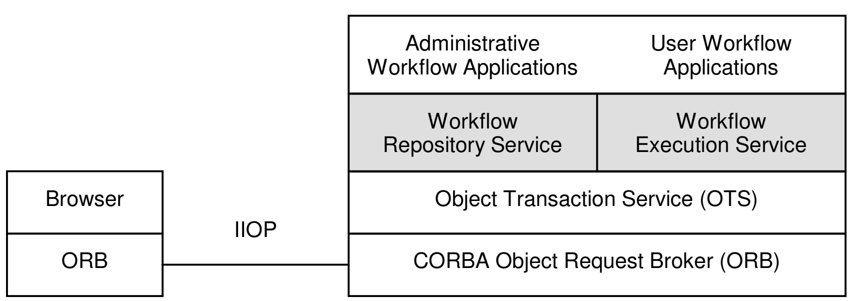

Fig. 2: Workflow management system structure.

Workflow Repository Service: The repository service stores workflow schemas and provides operations for initializing, modifying and inspecting schemas. A schema is represented according to the model briefly discussed below in terms of tasks, compound tasks, genesis tasks and dependencies. We have designed a scripting language that provides high-level notations (textual as well graphic) for the specification of schemas. The scripting language has been specifically designed to express task composition and inter-task dependencies of fault-tolerant distributed applications whose executions could span arbitrarily large durations [6]. We next highlight some of the features of the task model that enable flexible ways of composing an application:

- *Alternative input sources*: A task can acquire a given input from more than one source. This is the principal way of introducing redundant data sources for a task and for a task to control input selection.
- *Alternative outputs*: A task can terminate in one of several output states, producing distinct outcomes. Assume that a task is an atomic transaction that transfers a sum of money from customer account *A* to customer account *B* by debiting *A* and crediting *B*. Then one outcome could be the result of the task committing and the other outcome could an indication that the task has aborted.
- *Compound tasks*: A task can be composed from other tasks. This is the principal way of composing an application out of other applications. Individual tasks that make up an application can be *atomic* ('all or nothing' ACID transactions, possibly containing nested transactions within, with properties of: Atomicity, Consistency, Isolation and Durability) or *non-atomic*.
- *Genesis tasks*: A genesis task is a specialised form of compound task that acts as a placeholder for a task structure, represented as a workflow schema. Its main purpose is to enable dynamic (on demand) instantiation of that schema. Structuring an application in terms of genesis tasks then provides an efficient way of managing a very large workflow, as only those parts that are strictly needed are instantiated. Genesis tasks can also be utilised to specify workflow applications that contain recursive executions, that is a task structure whose execution will potentially cause the execution of its own structure as one of its sub-tasks.

A task is modelled as having a set of *input sets* and a set of *output sets*. In fig. 3, task t_i is represented as having two input sets I_1 and I_2, and two output sets O_1 and O_2. A task instance begins its life in a *wait* state, awaiting the availability of one of its input sets. The execution of a task is triggered (the state changes to *active*) by the availability of an input set, only the first available input set will trigger the task, the subsequent availability of other input sets will not trigger the task (if multiple input sets became available simultaneously, then the input set with the highest priority is chosen for processing). For an input set to be available it must have received all of its constituent input objects (i.e., indicating that all dataflow and notification dependencies have been satisfied). For example, in fig. 3, input set I_1 requires three dependencies to be satisfied: objects i_1 and i_2 must become available (dataflow dependencies) and one notification must be signalled (notifications are modelled as data-less input objects). A given input can be obtained from more than one source (e.g., three for i_3 in set I_2). If multiple sources of an input become available simultaneously, then the source with the highest priority is selected.

A task terminates (the state changes to *complete*) producing output objects belonging to exactly one of a set of output sets (O_1 or O_2 for task t_i). An output set consists of a (possibly empty) set of output objects (o_1 and o_2 for output set O_2).

Task instances manipulate references to input and output objects. A task is associated with one or more implementations (application code); at run time, a task instance is bound to a specific implementation.

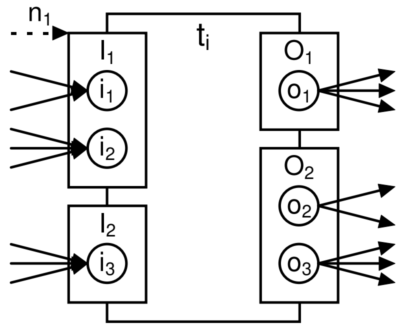

Fig. 3: A task

To allow applications to be recursively structured, the task model allows a task to be realized as a collection of tasks, this task is called a compound task. A task can either be a simple task (primitive task), a compound task (composed from other tasks) or a genesis task.

The workflow system stores workflow schemas in a repository and it is possible to modify the structure of a stored schema using transactions. For a complete specification of the repository service interfaces see [1].

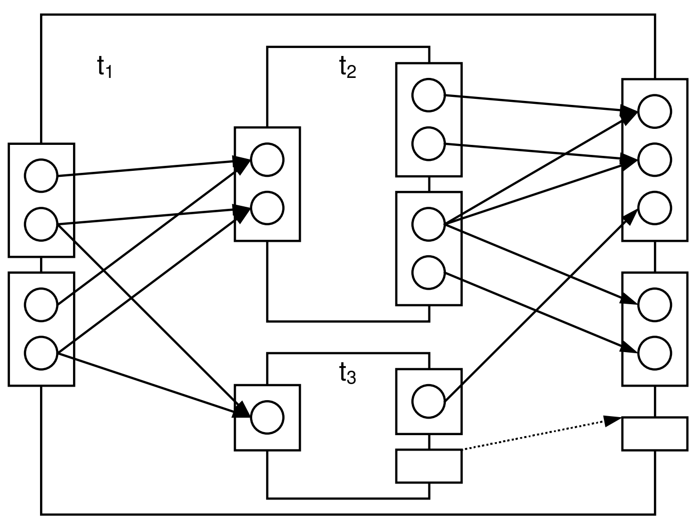

Fig. 4: A compound task.

Workflow Execution Service: The workflow execution service coordinates the execution of a workflow instance: it records inter-task dependencies of a schema in persistent atomic objects and uses atomic transactions for propagating coordination information to ensure that tasks are scheduled to run respecting their dependencies. The dependency information is maintained and managed by *task controllers*. Each task within a workflow application has a single dedicated task controller. The purpose of a task controller is to receive notifications of outputs of other task controllers and use this information to determine when its associated task can be started. The task controller is also responsible for propagating notifications of outputs of its task to other interested task controllers. Each task controller maintains a persistent, atomic object, `TaskControl` that is used for recording task dependencies. The structure is shown in fig. 5. For example, task controller tc_3 will co-ordinate with tc_1 and tc_2 to determine when t_3 can be started and propagate to tc_4 and tc_5 the results of t_3.

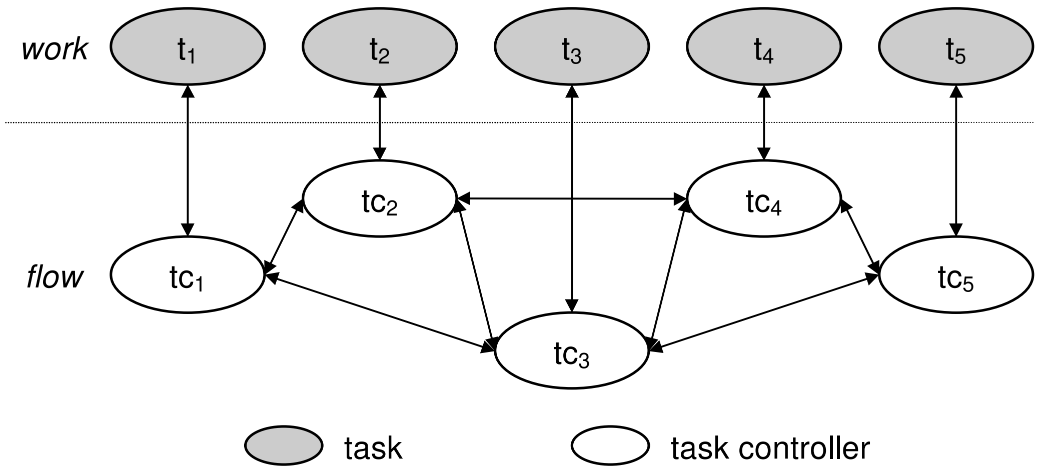

Fig. 5: Tasks and task controllers.

In addition to `TaskControl`, the workflow execution service maintains two other key objects: instances of `Resource`, and instances of transactional objects `Task`. Objects whose references are to be passed between workflow tasks are derived from `Resource`. `Task` objects represent the workflow tasks which make up a workflow application (`Tasks` are 'wrapper' objects to real application tasks). The most important operation contained within the `Task` interface is `start`, which takes as parameters: a reference to a `TaskControl` and a sequence of `Resource` references. The `TaskControl` reference is that of the controller of the task, and the sequence of `Resource` objects are the input parameters to the workflow task.

In summary, our system provides a very flexible and dependable task coordination facility that need not have any centralized control. Because the system is built using the underlying transactional layer, no additional recovery facilities are required for reliable task scheduling: provided failed nodes eventually recover and network partitions eventually heal, task notifications will be eventually completed. In addition, the system automatically records the application's execution history (audit trail): this can be derived from the committed states of `TaskControl` objects of the application. For a complete specification of the execution service interfaces, see [1].

Concluding Remarks

Currently available workflow systems are not scalable, as their structure tends to be monolithic. Further, they offer little support for building fault-tolerant distributed applications, nor can they inter-operate, as they make use of proprietary platforms and protocols. There is therefore much research activity on the construction of decentralized workflow architectures. System such as ours, RainMan [7], Coyote [8] and ORBWork from the METEOR research group [9] represent new generation of (research) systems that can work in arbitrarily distributed environments.

The task model and the transactional task coordination facility are the distinguishing features of our system and result in our system naturally providing a fault-tolerant ‘job scheduling’ environment that maintains a durable history of application interactions. Furthermore, in our system the execution service directly maintains the computation structure of an application within task controllers and makes it available through transactional operations; this makes the provision of system monitoring and dynamic workflows relatively easy. The wide-spread acceptance of CORBA and Java middleware technologies make our system ideally suited to building Internet applications. We are now working on building demonstrator applications in the area of electronic commerce.

Acknowledgements

This work has been supported in part by grants from Nortel Technology and Engineering and Physical Sciences Research Council, ESPRIT LTR Project C3DS (Project No. 24962) and ESPRIT Project MultiPLECX (Project No. 26810).

References

[1] Nortel and University of Newcastle upon Tyne, “Workflow Management Facility Specification”, OMG document bom/98-03-01, Submission for the OMG Business Object Domain Task Force (BODTF): Workflow Management Facility.

[2] F. Ranno, S.M. Wheater and S.K. Shrivastava, “A System for Specifying and Co-ordinating the Execution of Reliable Distributed Applications”, IFIP Conference on Distributed Applications and Interoperable Systems (DAIS’97), Cottbus, Germany, September 1997.

[3] S. M. Wheater, S. K. Shrivastava and F. Ranno, "A CORBA Compliant Transactional Workflow System for Internet Applications", Proc. of IFIP International Conf. on Distributed Systems Platforms (Middleware ‘98), September 15-18, 1998.

[4] S.K Shrivastava and S.M. Wheater, “Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications”, IEEE 4th Intl. Conf. on Configurable Distributed Systems, CDS’98, Annapolis, May 1998, pp. 10-17.

[5] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, “The Design and Implementation of Arjuna”, USENIX Computing Systems Journal, vol. 8 (3), pp. 255-308, Summer 1995.

[6] F. Ranno, S.K Shrivastava and S.M. Wheater, “A Language for Specifying the Composition of Reliable Distributed Applications”, 18th IEEE Intl. Conf. on Distributed Computing Systems, ICDCS’98, Amsterdam, May 1998, pp. 534-543.

[7] S. Paul, E. Park and J. Chaar, “RainMan: A Workflow System for the Internet”, Proc. of USENIX Symp. on Internet Technologies and Systems, November 1997.

[8] A. Dan and F. Parr, "The COYOTE Approach for Network Centric Service Applications: Conversational Service Transactions, a Monitor and an Application Style", Proc. of 7th Intl. Workshop on High Perf. Transaction Systems, Asilomar, California, Sept 1997, pp. 23-48.

[9] S. Das, K. Kochut, J. Miller, A. Seth and D. Worah, “ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for METEOR2”, Tech. Report No. UGA-CS-TR 97-001, Dept. of Computer Science, University of Georgia, Feb. 1997.