Reliability Mechanisms in the OPENflow Distributed Workflow System

Stuart M. Wheater and Santosh K. Shrivastava

Department of Computing Science,
University of Newcastle upon Tyne

Abstract: In a distributed environment, it is inevitable that long running applications will require support for fault-tolerance because, for example, machines may fail, networks may partition and services may be moved or withdrawn. This paper describes an approach that supports reliability in the domain of large scale distributed applications represented as business processes (commonly referred to as workflows). An application composition and execution environment has been designed and implemented as a transactional workflow system that enables sets of inter-related tasks (applications) to be carried out and supervised in a dependable manner. A task model that is expressive enough to capture application level exception handling by represent temporal dependencies between constituent tasks has been developed. Use of transactions ensures that changes can be carried out atomically with respect to running applications. The system serves as an example of the use of middleware technologies, based on CORBA, to provide a fault-tolerant execution environment for long running distributed applications.

1 Introduction

We consider applications in the Internet/Web environment. 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. Unfortunately, the Internet frequently suffers from failures which can affect both the performance and integrity of applications run over it.

Distributed objects plus ACID transactions provide a foundation for building high integrity applications. However, it has long been realised that ACID transactions by

themselves are not adequate for structuring long-lived applications [Gray 1981]. Top-level transactions are most suitably viewed as “short-lived” entities, performing stable state changes to the system. Long-lived top-level transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources for a long time; further, if such a transaction aborts, much valuable work already performed could be undone. In short, if an application is composed as a collection of transactions, then during run time, the entire activity representing the application in execution is frequently required to relax some of the ACID properties of the individual transactions. The entire activity can then be viewed as a non-ACID ‘extended transaction’. Much research work has been done on developing extended transaction models [e.g. Elmagarmid 1992]. Nevertheless, most of these techniques have not found any widespread usage, the main reason being lack of flexibility [Alonso et al. 1996]. As observed in [Alonso et al. 1996], transactional workflow systems with scripting facilities for expressing the composition of an activity (a business process) offer a flexible way of building application specific extended transactions.

Our approach is in the similar vain: we have implemented an application composition and execution environment as a transactional workflow system that enables sets of inter-related tasks to be carried out and supervised in a dependable manner. Currently available workflow systems are not scalable, as their structure tends to be monolithic. Further, they offer little support for building fault-tolerant applications, nor can they inter-operate, as they make use of proprietary platforms and protocols. Our system, called OPENflow, represents a significant departure from these; our system architecture is decentralized and open: it has been designed and implemented as a set of CORBA services, running on top of a given ORB. An overview of OPENflow appears in [Wheater *et al.* 1998]. This paper expands on fault tolerant aspects of our system. Our system architecture was the basis of Nortel’s submission to the OMG for a workflow standard [Nortel *et al.* 1998].

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.

In the next section we discuss requirements we wish to satisfy and the constrains on our design. The following section describes the task model supported by our workflow system. Section 4 then describes how the OPENflow system coordinates execution of distributed applications. Section 5 and 6 describes how system level and application level fault-tolerance has been implemented.

2 Requirements

In this section we will examine some of the requirements of a workflow system that is required to coordinate complex distributed applications reliably in a heterogeneous environment.

2.1 Fault-tolerance

Such applications will have a 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. The execution of such 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. It should be possible therefore to reconfigure an application dynamically because, for example, machines may fail, services may be moved or withdrawn and user requirements may change.

2.2 ORB independence

To allow the OPENflow to be used in a heterogeneous environment it is required to be relatively easy to port to new environments. Because OPENflow is CORBA based this may mean porting to a new ORB.

Because ORB vendors can and have made many different choices in the details of their ORB implementations, such as ORB functionality, achieving portability of CORBA applications and services is difficult. But by avoiding those ORB features that are non-standards or uncommon, the design of CORBA applications and services can be made fairly portable. Examples of such non-standard and uncommon CORBA features are:

- *Service recreations*: not all ORBs support creating/recreating of a service on a predefined object reference.
- *User information inclusion in object references*: This allows information to be passed about the object within the object reference, for example, a unique identify of the object or rebinding information. Unfortunately, not all ORBs support inclusion of user information in an object reference.
- *IIOP (Internet Inter-ORB Protocol) redirect*: this allows an invocation to an object to be redirected to an alternate if the original invocation fails. The IIOP redirect feature is not supported by all ORBs.
- *POA (Portable Object Adapter)*: this supports the complex management of object lifecycles. The POA support is currently not provided by all ORBs

By avoiding these features of ORBs the design of OPENflow has be made relatively ORB independent.

3 OPENflow Task Model

A task model must be expressive enough to be able to represent temporal dependencies of applications. The schema (task structure) represents a workflow application as a collection of tasks and their dependencies. A task is an application specific unit of activity that requires specified input objects and produces specified output objects. Dependency may be a notification dependency (indicating that a 'down stream' task can start only after a 'up stream' task has terminated (or started)) or a data-flow dependency (indicating that a 'down stream' task requires in addition to notification, input data from a 'up stream' task).

A task can be in one for three states: *wait, active* or *complete*; this is explained below. A task is modeled as having a set of *input sets* and a set of *output sets*. In figure 1, 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 inputs (i.e., indicating that all data-flow and notification inputs have been satisfied). For example, in figure 1, input set I_1 requires three inputs to be satisfied: two objects i_1 and i_2 must become available (data-flow dependencies) and one notification must be signalled (notifications are modeled 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 are available when the corresponding input set becomes available, 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_2 and o_3 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.

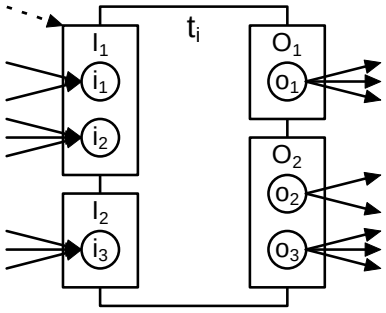

Figure 1, A task.

A schema indicates how the constituent tasks are 'connected'. We refer to the input alternative connecting a *source* object to a *sink* object. In figure 2 all the input alternatives of a task t_3 are labeled d_1, d_2, …, d_8. An example of an input (sink object) having multiple input alternatives is i_1, this has two input alternatives d_1 and d_2 (corresponding to two source object). Note that the source of an input alternative could be from an output set (e.g., d_4) or from an input set (e.g., d_7); the latter represents the case when an input is consumed by more than one task.

The notification dependencies are represented by dotted lines, for example, d_5 is a notification input alternative for n_1. The sink object of notification input alternative is a *virtual object* (n_1), and source object of notification input alternative corresponds to an "empty object" from an input or output set. An input set can contain multiple virtual objects, each corresponding to a separate temporal dependence which must be satisfied before the task can be started.

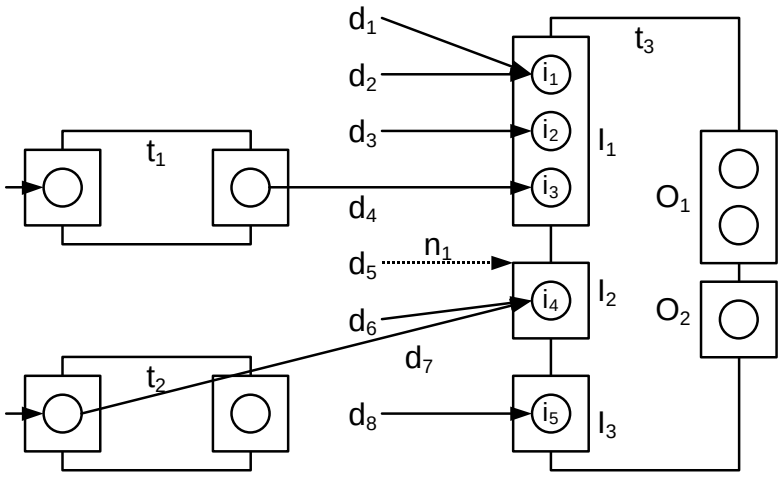

Figure 2, A workflow schema indicating inter-task dependencies.

A task within a workflow application can either be a *simple task* (primitive task which is indivisible), an *adapter task* (a specialised form of simple task for interoperability with

legacy workflow systems), a *compound task* or *genesis task*. The compound and genesis task forms will now be described in more details.

The task model allows a task to be realised from a collection of other tasks, such a task is called a compound task. A compound task undergoes the same state transitions as a simple task. The figure below illustrates a compound task, t_1, composed of tasks t_2 and t_3. A given output of a compound task can come from one or more internal sources (*output alternatives*). If multiple sources of an output become available simultaneously, then the source with the highest priority is selected.

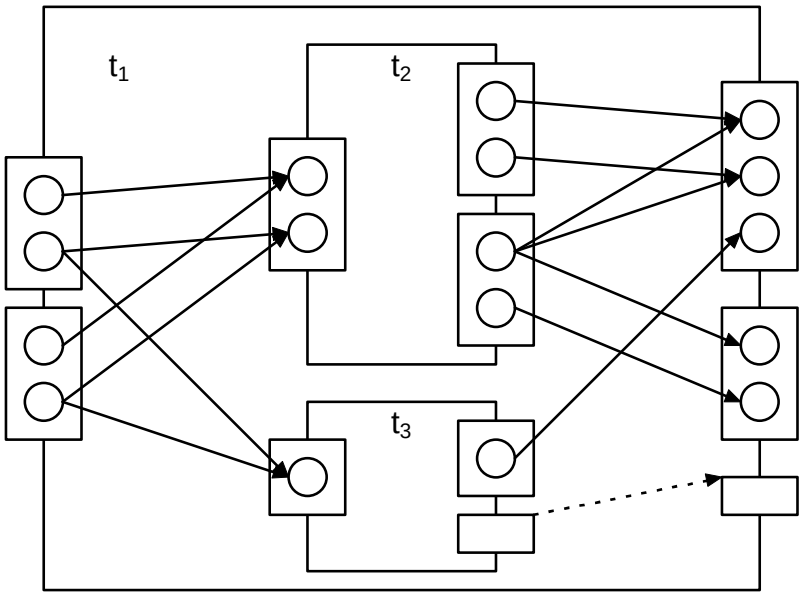

Figure 3, A compound task.

A genesis task can act as a place holder for a workflow schema (task structure), so enabling dynamic instantiation of that task structure, at the point when the genesis task starts. The figure below shows a genesis task, t_3 which is contained within a compound task t_1. If task t_2 terminates producing the upper output set, this will cause task t_3 to be started; t_3 being a genesis task, this will cause the instantiation of the schema associated with t_3.

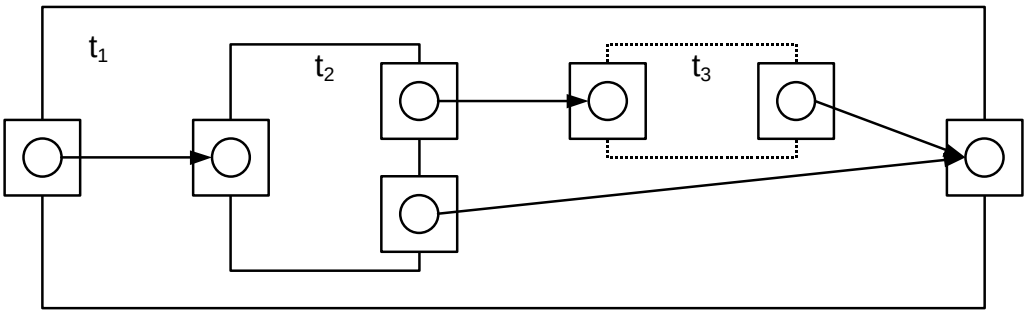

Figure 4, A genesis task.

A genesis task is used for on demand instantiation. For a complex application (modeled as a set of compound tasks) this enables instantiation of only those components which are strictly necessary. This also allows the execution of repetitive tasks.

Note that if the task structure associated with t_3 is the same task structure as the compound task t_1, this will produce a recursive structure resulting in the repeated execution of task t_2 (until task t_2 terminates producing the lower output set).

4 Distributed control

The OPENflow system structure is shown in figure 5. Here the box represents the software layers of the workflow system. 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 of the CORBA Object Transaction Service (OTS).

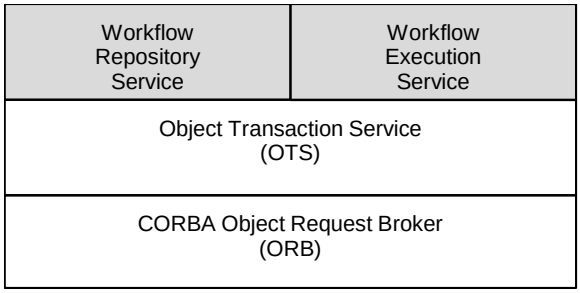

Figure 5, OPENflow system structure.

The workflow repository service stores workflow schemas and provides operations for initialising, modifying and inspecting schemas. A schema is represented according to the model briefly discussed earlier in terms of tasks, compound tasks, genesis tasks and dependencies.

The workflow execution service coordinates the execution of a workflow instance: it maintains the inter-task dependencies 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. When an input set is ready, the task controller notifies its task. When a task finishes, it notifies its controller, together with the output set produced. In this way a task controller maintains accurate information on the status of its task (waiting, active, complete). The task controller is also responsible for propagating notifications of outputs of its task to other interested task controllers. The structure is shown in figure 6. 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.

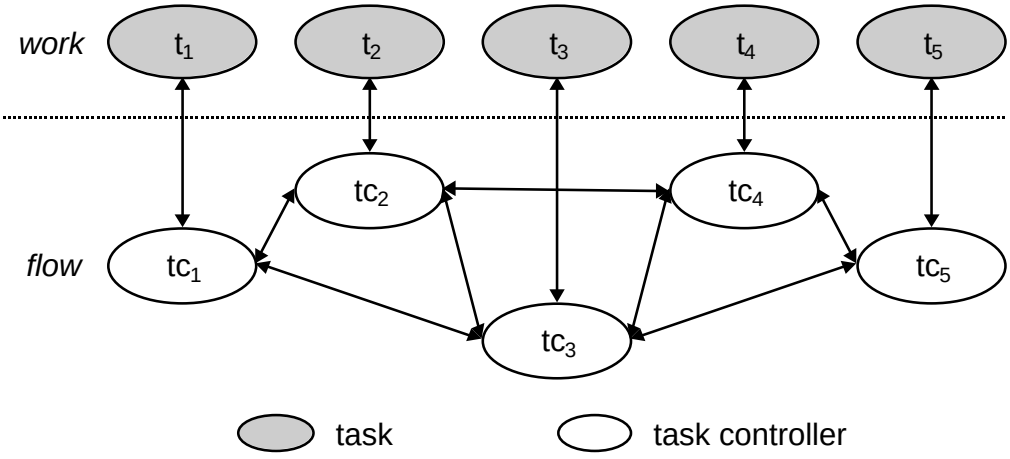

Figure 6, Tasks and task controllers.

The OPENflow system provides a very flexible task coordination facility that need not have any centralised control as in other systems, such as Rainman [Paul *et al.* 1997]. This means that the task controllers of an application can be grouped in an arbitrary manner. Figure 7 show a possible configuration.

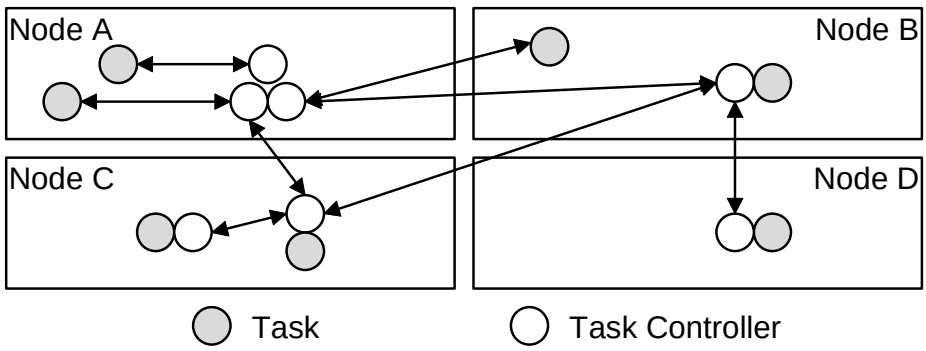

Figure 7, Task coordination.

5 System level fault-tolerance

The OPENflow system has been designed to tolerate a finite number communication and node failures. The main technical issues which needed to be addressed are how to design the sub-system that:

- Performs rebinding to services after their recovery after failure.
- Dependable notification of events between parts of the workflow, for example, an outcome between task controllers.

5.1 Object reference rebinding

In a CORBA application, services are accessed through object references. An object reference identifies a server and an object within that server, which is responsible for performing the operation of the service.

A major requirement, if continued progress of the application after recovery of a service is required, is to be able to regain access to the service after recovery from a failure.

One approach to this requirement would be to use persistent object references. When using persistent object references, services when they recover would require the same object references as the original services. But the use of persistent object references does present some problems:

- Not all ORBs support persistent object references.
- In general, a persistent object reference will contain the location (node) of the service that is accessible through the object reference. This means that service cannot be restarted at a different location (different node) than the original service. This would mean that the service cannot be made available until the machine that hosted the original service recovers.
- In general, a persistent object reference is not portable between ORBs. This means that service cannot be restarted on a different ORB than the original service.

Another approach is to use *IIOP (Internet Inter-ORB Protocol) redirect*. When using IIOP redirect the object reference for a service contains references to more that one object. If an invocation on an object fails, the invoker will reissue the invocation, and further invocations, to the next object. But the use of IIOP redirect does present some problems:

- Not all ORBs support IIOP redirect.
- The “alternate objects” need to be set-up and specified before invocation.

Because of these problems OPENflow was designed to use different approach. The OPENflow approach is as follows: all services are registered in the naming service and all services provide an operation, called *id*, that returns the "name" that they are registered under. This means that when a service recovers it is required to update its entry in the naming service. Through the naming service, user of the service can then obtain the updated entry and use it to regain access the service.

Because it is the responsibility of the service to reregister itself with the naming service, when it recovers, the naming service is not required to be fault-tolerant, or even have a persistent state. Also, because the naming service has been designed to be a federated service, it does not represent a single point of failure or performance bottleneck.

To illustrate the design of the OPENflow object reference rebinding mechanism the three main scenarios will now be described:

The first scenario, illustrated in figure 8, is when a client invokes an operation (op) of a service (x). If an invocation by the client on the service fails, because of lack of response, the client contacts the naming service to obtain the latest object reference to the service. If the object reference to the service has changed the new object reference is used to reissue the invocation, otherwise an exception is raised to indicate that the operation has failed. This operation can be retried later at the client discretion.

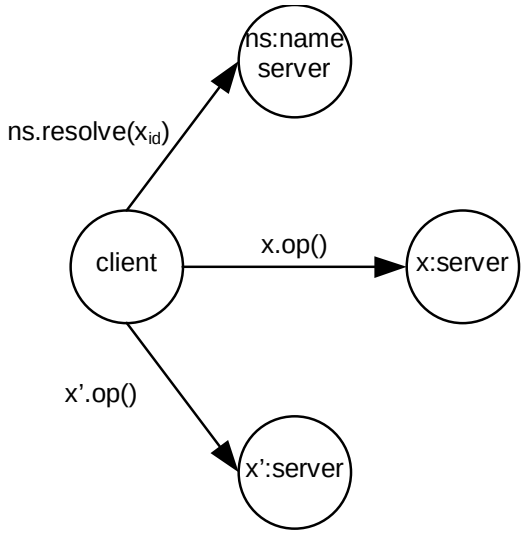

Figure 8, operation on x.

To ensure that the rebinding mechanism is efficient all the object references to a service within the client should be replaced in one go. This can be achieved by maintaining objects within the client that act as a proxy for the service. This proxy will perform the rebinding if any operation on the service fails.

The second scenario, illustrated in figure 9, is when a client invokes an operation on a service (y) and passing an object reference to another service (x) as a parameter. If the target service (y) of the invocation is available, the invoked operation will first obtain the "name" associated with the service (x) that is passed as a parameter, this is done by invoking the *id* operation on the service. The name will allow the server (x) to rebind to that server (y) if it fails. If the client has unknowingly passed an invalid object reference to *x* then the server *y* will not be able to obtain the "name" associated with *x* because the *id* operation will fail, in this case the operation would raise an exception to indicate this to the client, then the client would rebind its object reference to *x* and reissue the invocation. The object that act as a proxy within the client would perform this rebinding, this reduces the complexity if the client code, by putting all the rebinding in a single piece of code rather that with each service call in the client.

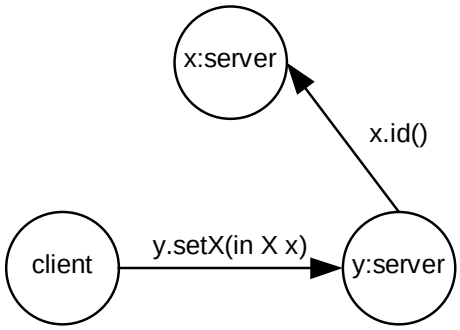

Figure 9, set operation on y with x as parameter.

To ensure that this process is efficient the servers (and clients) contain a cache of services to "name" mappings. This means that if a service has been used recently the service will not need to be contacted to obtain its "name".

The third scenario, illustrated in figure 10, is when a client invokes an operation on a service (y) that returns an object reference to another service (x). This scenario is the most complex to address. If the target service *y* of the invocation is available, the invoked operation will return an object reference to service *x*. If this object reference is valid the client will be able to obtain the service's (x) "name". On the other hand if the object reference return from the service (y) is invalid, then the object reference can not be used to obtain the "name" of the service.

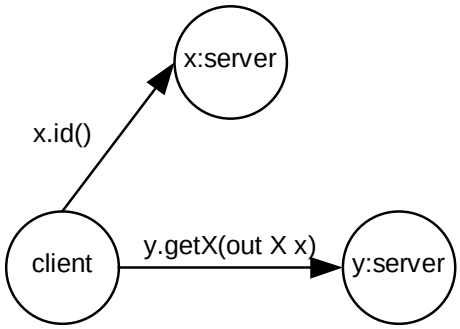

Figure 10, get operation on y with x as parameter.

This scenario is complicated because it is the client that discovers that the service (y) is holding an invalid object reference to service (x). To be able to make continued progress the object reference within the service (y) must be rebound. Two simple approaches which could be performed in the server are:

- To rebind all object references returned from a service just before they are returned.
- To periodically rebind all object references within a server.

Both these approaches would cause a large number of valid object references to be rebound. In circumstances where there are only a limited number of failures this would be inefficient, so another approach is used.

The approach used within OPENflow is that each server also provides a service that can cause rebinding of object references within the server. This service is registered with the name server under a name that can be deduced from the names of the other services provided by the server. This service will be used by a client which receives an invalid object reference from a server, the client can cause the object reference to be rebound before the client reissues its invocation.

Following this pattern of interaction will ensure that object references can be rebound after their associated services fail, so ensuring continued progress after failures. This approach is relatively ORB independent, in that it only required a naming service, which is commonly provided with most ORBs. Also it only requires a slight modification to the interfaces to services, the addition of the *id* operation. Although the implementation of the service proxies is complicated, after that there are only relatively small modifications required to the code of the clients and services.

5.2 Dependable notifications

To ensure that the coordination of the application is performed reliably, the notification used for coordination need to be reliable. The task controllers are responsible for propagating notifications of outputs of its task to other interested task controllers. Each task controller maintains a persistent, atomic object that is used for recording task

dependencies. The use of distributed transactions ensures that both client and server have a consistent view on whether the notification has been performed, or not.

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 workflow's execution history (audit trail): this is maintained in the committed states of the task controllers of the workflow.

6 Application level fault-tolerance

6.1 Flexible task composition

The task model descried earlier has been constructed with fault-tolerance in mind. It enables an application builder to incorporate alternative tasks, compensating tasks, replacement tasks etc., within an application to deal with a variety of exceptional situations. Individual tasks that make up an application can be *atomic* ('all or nothing' top level ACID transactions, possibly containing nested transactions within, with properties of: Atomicity, Consistency, Isolation and Durability) or *non-atomic*. The mechanisms described below can be used for constructing application specific extended transactions including Sagas [Garcia-Molina 1987].

6.1.1 Alternate sources of input

One common required feature is to be able to acquire an input from a number of alternate sources, this feature is supported by the task model, and is illustrated in figure 11. In figure 11, the input required by task *D* can be received from either task *A*, *B* or *C* as a result of their completion.

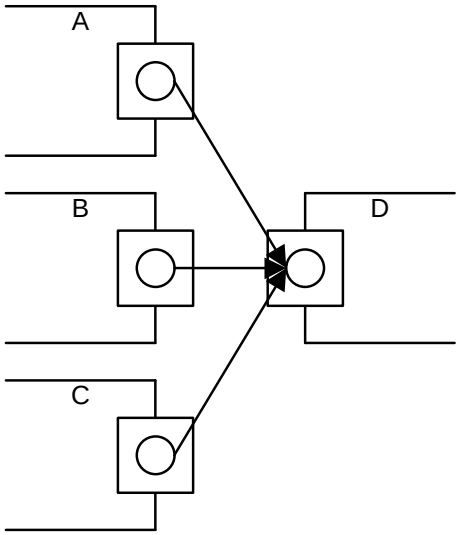

Figure 11, Example of an alternate sources of input.

This feature allows an application designer to construct more complicated task structures, example of some specific structures are given in the following sections.

6.1.2 Support for alternate tasks

A commonly required pattern for fault–tolerance is the using of a secondary (alternate) if the primary fails. This can also be done with tasks within an application. The use of an alternate task is illustrated in figure 12. In this example a task *A* has been provided with an alternate *A'*. The task structure is arranged so as that input selected by task *A* is provided to task *A'*. If task *A* is successful, its outcome will be the upper of its output sets, which will cause task *B* to start with task *A* results. On the other hand if task *A* fails, its outcome will be the lower of its output sets, which will cause task *A'*, its alternate, to be started. If task *A'* is successful it will cause task *B* to start. If task *A'* on the other hand fails it can trigger some higher level fault-tolerance or cause the clean termination of the application.

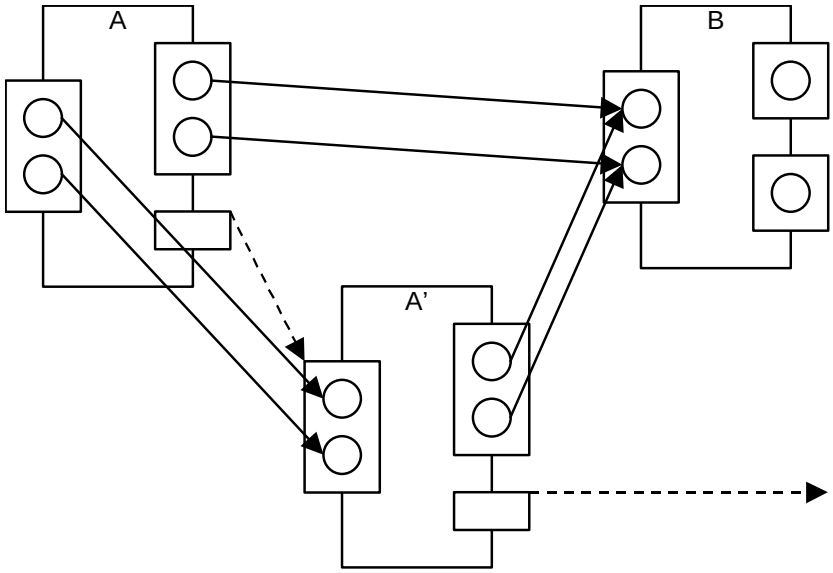

Figure 12, Example of an alternate task.

6.1.2 Support for forward/backward error recovery

Forward error recovery attempts to “move the system state forward” to a consistent state by performing additional operations (compensation). Whereas backward error recovery attempts to “move the system state backward” to a consistent state by performing additional operations which undo the effects of other operations.

Examples of the task structuring required for forward and backward error recovery are illustrated in figure 13 and 14. In both task structures task *A* can complete in three ways: “totally successful” (upper output set), “partially successful” (middle output set) and “with no effects” (bottom output set). The “totally successful” and “with no effects” outcomes of task *A* will trigger corresponding input sets of task *B*. The “partially successful” outcome of task *A* will either trigger the task *A*+, for forward error recovery, and task *A*-, for backward error recovery. If successful the task *A*+ will cause “totally successful” input set of task *B* to be triggered. If successful the task *A*- will cause the “with no effects” input set of task *B* to be triggered. If either task *A*+ or *A*- is unsuccessful it can trigger some higher level fault-tolerance or cause the clean termination of the application.

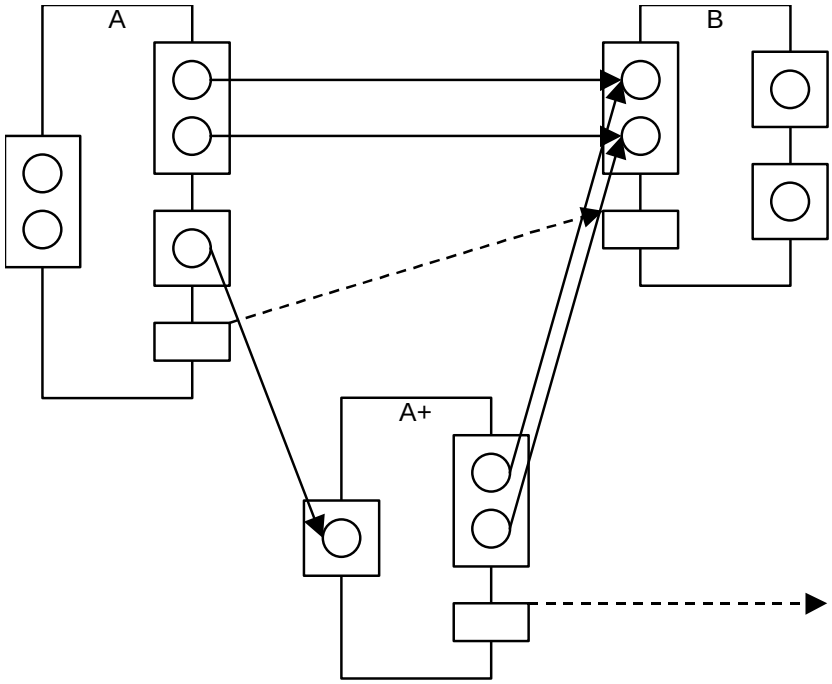

Figure 13, Example of a forward error recovery task.

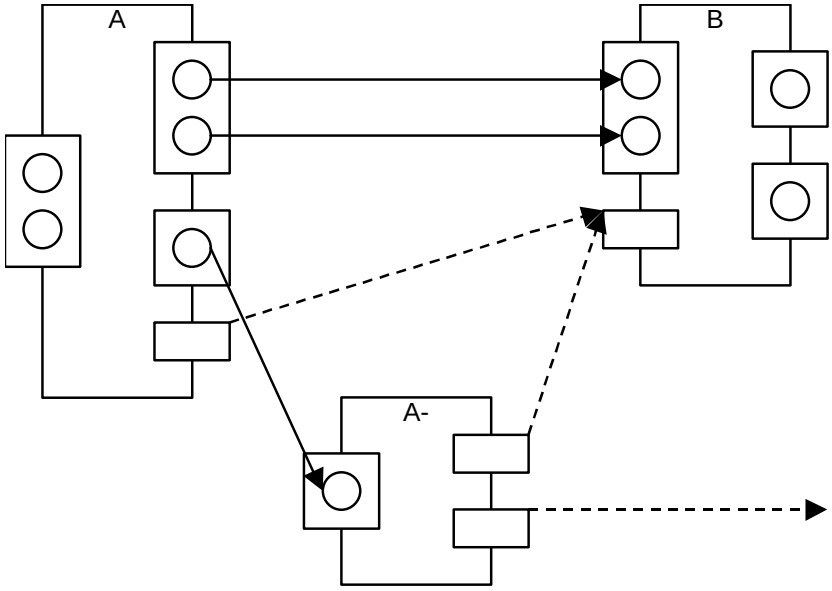

Figure 14, Example of a backward error recovery task.

6.1.4 Support for input time-out

A commonly required feature is not to have to block for an input ‘for ever’. The task model is able to support the “timing-out of inputs”, so a task structure can be specified which will mean that a task can be triggered if the task which it depends upon does not

complete within a fixed period of time. Such a task structure is illustrated in figure 15. In figure 15, if the task *A* does not complete within a specified period of time task *TO* will trigger the starting of task *B*.

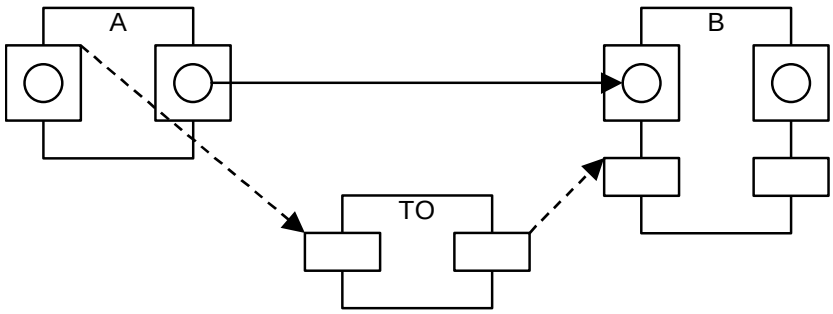

Figure 15, Example of an input time-out task.

6.2 Dynamic reconfiguration support

The OPENflow system achieves its support for dynamic reconfiguration thought *reflection*. The system is reflective, as it maintains its structure as information, and makes it available through transactional operations, this allows changes to be made (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*) [Shrivastava and Wheater 1998]. The use of transactions ensures that changes to schemas and instances are carried out atomically with respect to normal processing. Below is the list of possible changes that can be performed on a workflow instance:

- The implementation bound to a simple task can be changed.
- Tasks can be added or removed from workflow instances.
- The constituent tasks of a compound task can be changed.
- Input alternatives can be added and removed from a task.
- The priority associated with input alternatives of a task can be changed.
- Output alternatives can be added and removed from a compound task.
- The priority associated with output alternatives of a compound task can be changed.
- The task structure associated with a genesis task can be changed.

These changes must be performed consistently, by which we mean respecting two conditions: (i) modifications to a workflow schema instance are carried out atomically

(either all changes are performed or none) with respect to the normal processing activities; and (ii) the application is able to execute respecting these changes. We use transactions to respect condition one. In addition, the following restrictions need to be observed to respect condition two.

R1: The implementation bound to a simple task can be changed, provided the task is in the wait state.

R2: The task structure bound to a genesis task can be changed, provided the task is in the wait state.

R3: Input alternatives cannot be added, removed or in anyway modified for tasks that are in state *active* or *complete*.

R4: Output alternatives cannot be added, removed or in anyway modified for a compound task that is in *complete* state

7 Conclusions

We have described reliability mechanisms of OPENflow, an application composition and execution environment that enables sets of inter-related tasks to be carried out and supervised in a dependable manner. The system meets the requirements of interoperability, scalability, flexible task composition, dependability and dynamic reconfiguration. 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.

8 Acknowledgements

This work has been supported in part by grants from ESPRIT LTR Project C3DS (Project No. 24962), ESPRIT Project MultiPLECX (Project No. 26810) and Nortel Networks. Support from colleagues Frédéric Ranno and Jonathan Halliday at Newcastle Univerisity and Samantha Merrion, Dave Stringer, Harold Toze and John Warne at Nortel Technology, Harlow is gratefully acknowledged.

References

[Gray 1981] J. N. Gray, “The transaction concept: virtues and limitations”, Proceedings of the 7th VLDB Conference, September 1981, pp. 144-154.

[Elmagarmid 1992] A. K. Elmagarmid (ed), “Transaction models for advanced database applications”, Morgan Kaufmann, 1992.

[Alonso et al. 1996] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan, “Advanced transaction models in workflow contexts”, Proc. of 12th Intl. Conf. on Data Engineering, New Orleans, March 1996.

[Wheater *et al.* 1998] S.M. Wheater, S.K. Shrivastava and F. Ranno, “A CORBA Compliant Transactional Workflow System for Internet Applications”, Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE’98), The Lake District, England, pp. 3-18, September 1998.

[Nortel *et al.* 1998] Nortel and University of Newcastle upon Tyne, “Workflow Management Facility Specification”, OMG document number bom/98-03-01, March 1998.

[Paul *et al.* 1997] 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.

[Shrivastava and Wheater 1998] S K Shrivastava and S M Wheater, “Architectural Support for Dynamic Reconfiguration of distributed workflow Applications”, IEE Proceedings – Software, Vol. 145, No. 5, October 1998, pp. 155-162.

[Garcia-Molina 1987] H. Garcia-Molina and K. Salem, “Sagas”, Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1987.