Enhancing workflow fault tolerance Using Replication Technique

by

Xiaoliang Zhu

(Under the direction of Krzysztof J. Kochut)

Abstract

ORBWork is a scalable distributed runtime system of the METEOR workflow management system. This thesis investigates providing fault tolerance through replication in workflow systems. The passive and active replication models are implemented using Sun Jini technology and applied to ORBWork. Our experiments show that replication can greatly enhance the fault tolerance of the workflow system. Especially, the system can tolerate fatal failures including server crashes.

Index words: Workflow, Replica, Fault Tolerance, Computer, Reliability, Availability, Exception, Java, RMI, Jini

Enhancing workflow fault tolerance Using Replication Technique

by

Xiaoliang Zhu

B.S., The University of Science and Technology of China, China, 1998

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2002

Enhancing workflow fault tolerance Using Replication Technique

by

Xiaoliang Zhu

Approved:

Major Professor: Krzysztof J. Kochut

Committee: Amit P. Sheth
Daniel M. Everett

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
August 2002

Acknowledgements

I am extremely thankful to my major professor, Dr. Krzysztof Kochut for his invaluable guidance and encouragement throughout my entire research and academic years. I would also like to thank Dr. Amit P. Sheth and Dr. Daniel M. Everett for serving on my advisory committee, and for their advice and guidance. I also apprepriate Dr. Robinson for serving on my advisory committee although he could not attend my final defense due to schedule conflict.

My sincere gratitude is extended to my wonderful and loving parents, who have been very supportive and encouraging. My friends, including but not limited to, Lirong Cheng, Albert Fu, Yangrong Ling, Cejun Liu, Guangliang Pan, Feng Sun, Fugao Wang, Qun Wang, Xuemei Wang, Yunzhou Wu, Hanjin Yan, Jie Zhang, Weicheng Zhang, Wenduo Zhou, have also been very helpful to me both in life and in research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A workflow is an activity involving the coordinated execution of multiple tasks performed by different processing entities [KS95]. A Workflow Management System (WFMS) is a set of tools providing support for the necessary services of workflow creation (which includes process definition), workflow enactment, and administration and monitoring of workflow processes [SK98]. Typical workflow examples include billing and loan approval in financial corporations, claim processing in insurance companies, and patient admittance in a health care organization [SWK97].

The current state-of-the-art in WFMSs is dictated by the commercial market [SK98]. Today's business enterprise must deal with global competition, reduce cost, and rapidly develop new services and products. To address these requirements enterprises must constantly reconsider and optimize the way they do business and change their information systems and applications to support evolving business processes. Workflow technology facilitates these by providing methodologies and software to model business process as workflow specifications, reengineer business process to optimize specified processes, and automatically generate workflow implementations from workflow specifications [GHS95]. The demand for workflow systems has grown so rapidly that in just a few years several hundred products have appeared in the market [AH00, GHS95].

To increase the availability, scalability and reliability, people have turned to distributed systems to decentralize workflow systems. This decentralization is also justified by the fact that the information resources of many modern corporations can

best be characterized as a collection of widely heterogeneous, largely distributed, and loosely coupled computing environments. Many current trends reinforce this characterization: the decentralization of the corporation and of decision making, the emphasis on client-server architectures, the relevance of federated systems, and the increasing availability of distributed-processing technologies such as the Web, CORBA, Object Linking and Embedding, Java, Video Conference, to mention a few. All these trends promote the deployment of large and heterogeneous distributed execution environments where interrelated tasks can be efficiently executed and closely monitored [Alo00].

However, despite the overall success in workflow systems, fault-tolerance has been a challenging issue. Most existing commercial workflow products lack the redundancy and flexibility to replace failed components [AH00], and have poor performance in case of failures, such as network partition and host crashes. A single point of failure can bring the whole system down, and there are no mechanisms for keeping the system functioning without interruption. Much research has been done in the area of exception/failure handling [DKM96, Wor97, WSR98, HA98, Koc98, MOK99, LSK00]. Most of the research concentrated on introducing the transactional concepts into workflow management [DKM96, WSR98, MOK99], or applying the combination of database transaction concepts and exception signal/handler concepts in programming languages to workflow systems [HA98, Koc98]. Another effort is to handle task exceptions within a more precise context [Wor97, LSK00]. However, the issue of providing redundancy in workflow systems has not been addressed adequately. The IBM Exotica project provides high availability through replication [KAG95, MAG95], but it mainly replicates the underlying databases, instead of workflow components. This thesis is dedicated to providing redundancy and adding fault tolerance to workflow systems. In particular, our contributions include:

- Creating a framework for supplying fault tolerance to a workflow system.
- Developing a replication package that is reusable for general replication purposes.
- Implementing fault tolerance as an enhancement to METEOR/ORBWork.

The rest of the thesis is organized as follows. Chapter 2 presents an overview of WFMS, especially the architecture and implementation of METEOR/ORBWork in our lab. In Chapter 3, we give background on fault tolerance in distributed systems, as METEOR/ORBWork is also classified as a distributed system. In Chapter 4, we describe how fault tolerance was provided in our METEOR/ORBWork system ($\text{ORBWork}_{\text{FT}}$). Chapter 5 gives detailed algorithms implementing replication models. Chapter 6 contains description of a series of experiments to test the fault tolerance of the $\text{ORBWork}_{\text{FT}}$. Conclusions and future work are provided in Chapter 7.

Chapter 2

Workflow Systems

2.1 Introduction

Every business organization has some procedures which need to be followed to provide efficiency, consistency and quality in the entire work process. Workflow management systems are used to coordinate and streamline business processes. However, there is little agreement as to what workflow is and which features a workflow management system must provide [GHS95]. Under the umbrella of the term "workflow", which is often used casually, people may be referring to a business process, specification of a process, software that implements and automates a process, or software that simply supports the coordination and collaboration of people that implement a process [GHS95]. The variety in workflow definition has leads to poor interoperability among workflow products. Many efforts have been done to promote interoperability of workflow management systems. The Workflow Management Coalition (WfMC) provided a workflow reference model [WMC95]. Aalst et al. addressed systematically workflow requirements and concluded 26 workflow patterns [ABH00].

In general, workflow tools represent business processes as workflow processes — that is, as computerized models of the business process where all aspects necessary for executing the process are specified. These aspects include defining each individual step in the process, the order and conditions in which the steps must be executed, the data flow between steps, identifying who is responsible for each step. These steps are called tasks. A typical business process is then comprised of multiple cooperating

tasks. A WFMS can thus be seen as the set of tools used to design and define workflow processes, the environment in which these processes are executed, and the corresponding set of interfaces to the users and applications [AH00]. In other words, WFMS provides support in the areas of process definition, workflow enactment, administration and monitoring of workflow processes [Hol94].

The process model describes the structure of the business process in the real world. Each business process has different paths that need to be taken based on different rules. The process model is responsible for describing all the possible paths in a business process along with the different rules that lead to the different paths and also details out the necessary actions that need to be performed in each path. Process instances are created based on the defined process model. Each instance is a single thread of execution of the process.

Any process consists of a number of steps that contribute toward the completion of a process. An activity is a logical step or description of such a piece of work. It could be either a manual process like filling out a form, or an automated task like updating the records in a database. A workflow is a process consisting of a collection of activities across time and space. It provides a framework for the integration and coordination of distributed resources, tasks and individuals.

A task is considered to be the smallest atomic unit of a workflow, representing some kind of work to be done. Tasks can be categorized as manual, which are performed by a human, or automated, which are done by programs. The execution of tasks by different processing entities can be controlled by a human coordinator or can be automated by *Workflow Management System (WFMS).*

2.2 ORBWork/METEOR Architecture

The METEOR project is an ongoing workflow system in the LSDIS Laboratory of the University of Georgia. Its objective is to support and enable automated solutions for enterprise applications. METEOR provides an open system based high-end workflow management solution along with an enterprise application integration infrastructure [KSM99]. It also provides a very well defined model and language for specifying the task details in a workflow, compiling the details, and controlling the execution of workflow processes in a distributed environment. METEOR's architecture comprises of a collection of services, implemented as separate modules: design and modeling tools, workflow repository and the enactment system, as shown in Figure 2.1. The repository service is responsible for maintaining information about workflow definitions and associated workflow applications. The workflow designer, a graphical design tool, communicate with repository service and retrieve, updates, and stores workflow definitions. It is also capable of browsing the contents of the repository. The repository service is also available to the enactment service and provides the necessary information about a workflow application to be started. The enactment service provides the necessary functionality for running workflow instances, and is subdivided into scheduling and monitoring. Two different enactment services are provided by METEOR: ORBWork and WebWork. In this thesis, we only concentrate on ORBWork. ORBWork has been implemented to support the execution of workflows in a heterogeneous, autonomous and distributed (HAD) environment. ORBWork consists of task schedulers, task managers, monitor, ORBWork manager, ORBWork server, and data servers, as shown in Figure 2.2. Figure 2.3 shows the interaction among these components [Tri00]. A more detailed description follows.

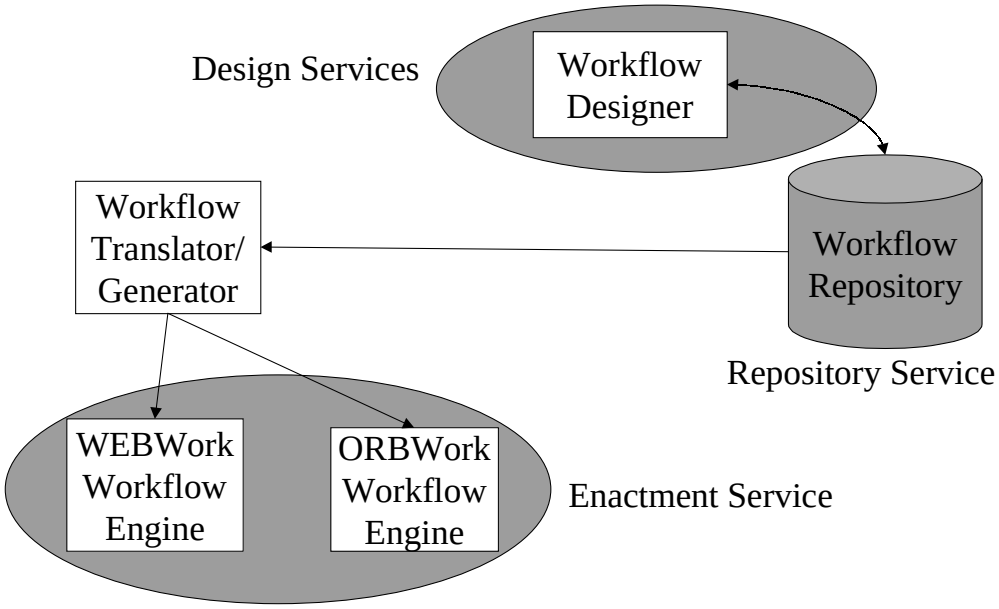

Figure 2.1: METEOR Architecture

2.2.1 Monitor

The *monitor* is extensively used by other ORBWork components to log all kinds of event messages. It provides a user interface for the workflow administrator to access the log information about the workflow system.

2.2.2 Workflow Manager

The *workflow manager* is used to install new workflow processes (schemata), modify the existing processes, and keep track of the activities of the scheduler. Other ORB-Work components register themselves with the Workflow manager. It implements a small subset of the HTTP protocol, and thus implements a light weight local Web

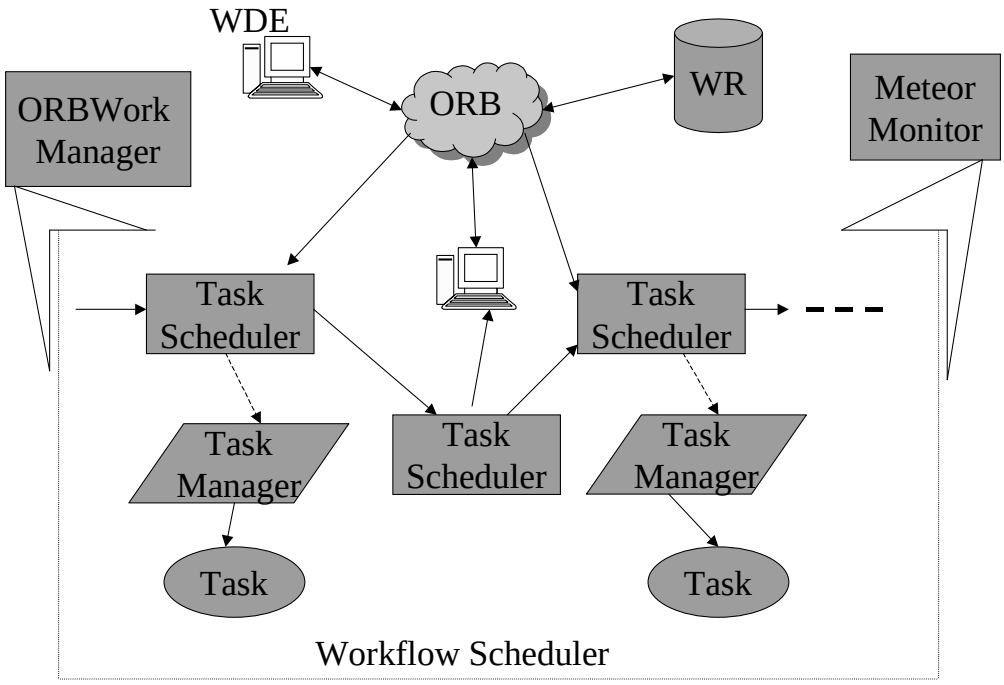

Figure 2.2: ORBWork System

Server. This enables users and administrators to interact with it. In fact, users always connect to the ORBWork manager as the first step to access the workflow system.

2.2.3 ORBWork Scheduler

The scheduling information in ORBWork is distributed in a number of schedulers. One scheduler is created for each activity/task involved in the workflow process. Thus, ORBWork's scheduler is composed of a number of small schedulers. The individual schedulers are called task schedulers. Task scheduler accesses task specifications directly from the repository through the HTTP protocol. The task specification consists of input transitions, output transitions with associated conditions and data objects sent into and out of the task. Like workflow managers, Task schedulers also

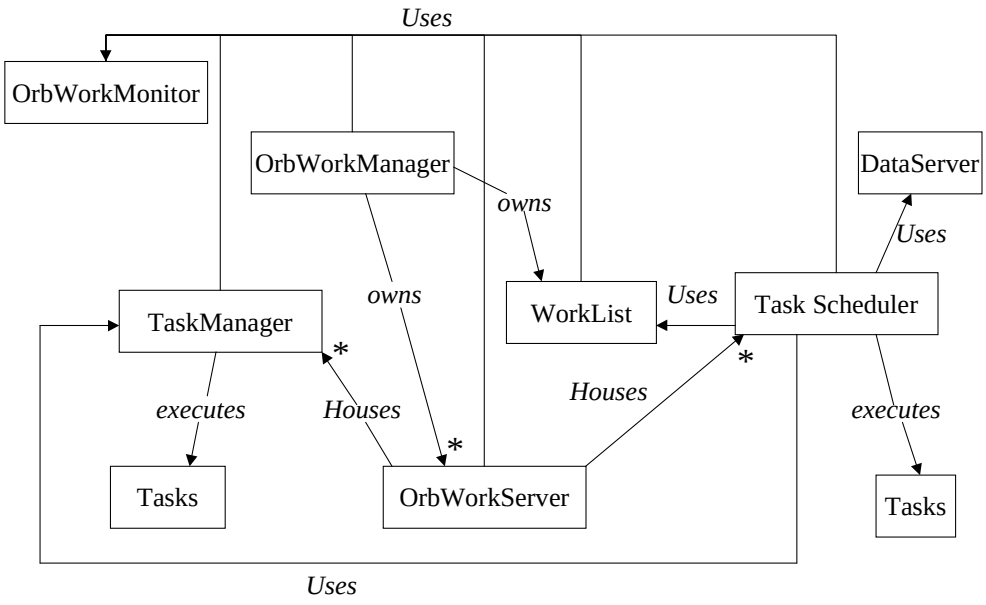

Figure 2.3: Interaction among ORBWork Components

implement a small subset of the HTTP protocol, and thus implement a light weight local Web Server. This enables an ORBWork administrator to interact with a particular task scheduler from a common Web browser, and make necessary changes and modifications to the scheduling information.

2.2.4 Task Manager

The enactment service also contains task managers. Task Managers control the execution of all non-human tasks. Based on the task type, as a workflow instance progresses through its execution, individual task schedulers create appropriate task managers to oversee the execution of associated tasks, unless the task (human task) has been designed to have a worklist.

2.2.5 Data Server

The data server houses the data objects necessary for the execution of any workflow instance. The data objects are dynamically created and made available to the successor tasks. Instead of passing the whole data object, only a reference to it is sent to the task scheduler needing it. When the task scheduler is ready to run the task, it accesses the necessary data object and extracts the relevant attribute values.

2.2.6 ORBWork Server

The ORBWork server houses the various task schedulers. As this is a distributed system, there can exist many ORBWork servers residing on different hosts. Each of the servers is designed to hold and control a number of task schedulers.

2.2.7 Worklist

All the human task instances in the workflow reside on the *worklist*. The worklist is a lightweight HTTP Server and provides a well-constrained subset of the HTTP protocol. This allows the end users to interact directly with all the human tasks of a particular workflow instance that resides in the worklist through a common web browser.

2.3 Workflow Transport Protocol

To facilitate the interaction among ORBWork components, Workflow Transport Protocol (WFTP) has been developed so that users and clients do not have to concern the implementation details when they use ORBWork services. WFTP also provides interfaces to interoperate with other workflow enactment services. WFTP interfaces give developers great flexibility to change and improve the internal implementation without affecting clients' code. In this thesis, the WFTP framework also facilitates

the implementation of fault tolerance. The WFTP architecture is shown in Figure 2.4.

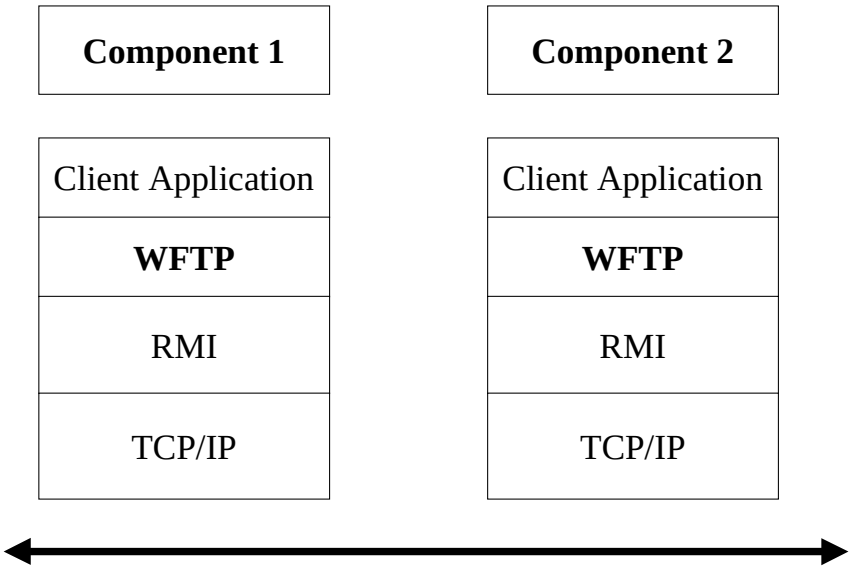

Figure 2.4: Architecture of the System. Each component implements a WFTP interface

With such an architecture, the method calls are handled as in Figure 2.5. As we can see, clients only need to interact with WFTP interfaces. The implementation of WFTP then invokes the corresponding RMI implementation which in turn delegates to the actual object. The WFTP interfaces act as the front ends for ORBWork services. Notice that the RMI here refers to general remote method invocation, not necessarily Java RMI.

2.4 The Java RMI Implementation of ORBWork

ORBWork has been implemented using Java RMI technology [Tri00]. Here we will have a brief overview of this implementation.

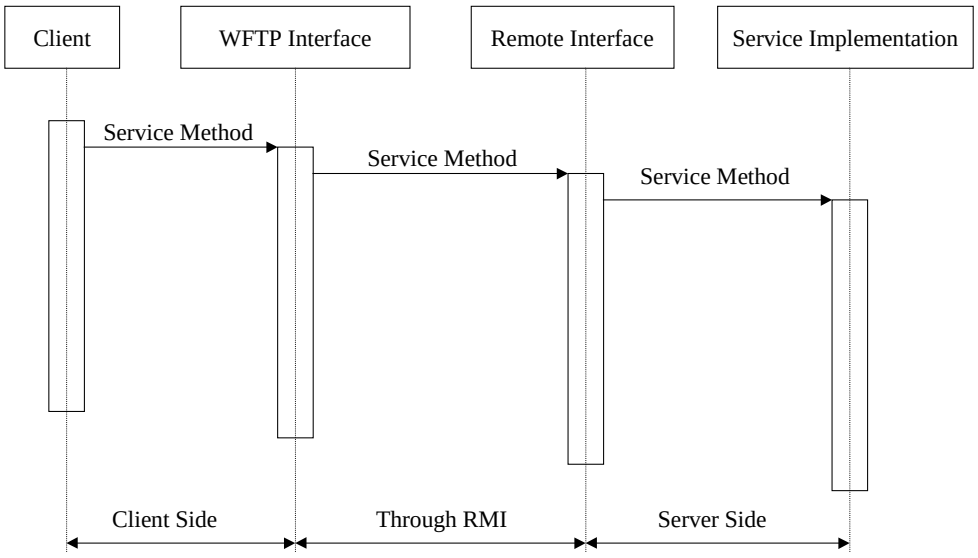

Figure 2.5: Sequence Diagram of Method Call

2.4.1 Introduction to Java RMI

The Java RMI [RMI] architecture is based on the separability between interface and implementation. RMI allows the code that defines an interface to be separated from the one that implements that interface and run on a separate JVM. Java RMI works in a similar fashion as CORBA in that it supplies a transport layer to handle the details of data transmission (marshaling and unmarshaling), encoding and call protocols. The server object implements functions and makes them available to clients through its interface. The server object must be exported via a server application. Clients obtain a reference to the server object, either by looking up the object by name with a registry if the server object has been registered, or by receiving the reference as an argument or a return value. The reference is not the

ORBWorkManagerIf	A remote interface
RMIORBWorkManagerImpl	A server class implementing the remote interface
ORBWorkMgrServer	A server program that creates server objects and binds them to the registry
WFTP_ORBWorkManagerIf	The interface to the clients
WFTP_ORBWorkManagerImpl	A class implementing WFTP interface, the client to the remote class
RMIORBWorkMgrImpl_Stub	A stub class that is automatically generated by the rmic program
RMIORBWorkMgrImpl_Skel	A skeleton class that is automatically generated by the rmic program

Table 2.1: Naming Convention for ORBWorkManager

actual server object, but a proxy or stub to the actual server object. Clients invoke the proxy object which delegates the invocation to the actual server object.

The RMI registry is a naming service that maps names to registered remote objects. A server binds its remote object with a name in the registry, and a client can then get a reference to the remote object by looking up the service name in the registry. The registry needs to be started before any objects are bound to it. And for security reason, a server can only bind its remote object in a registry on a local host, although it can do a lookup upon registries on any host. We will see later in this thesis that this requirement no longer exists in Jini.

2.4.2 Implementation

A set of classes come into play in the RMI implementation. It is important to follow a standard naming convention to make the implementation easier to understand. Table 2.1 gives the naming convention we have used for the Manager component. All the other components follow the similar naming convention [Tri00].

Below are some main issues when defining and implementing the remote interfaces.

- The interface must be declared public.
- The interface must extend java.rmi.Remote interface.
- Each method in the remote interface must declare java.rmi.RemoteException in its throws clause.
- The implementation must implement a remote interface that extends java.rmi.Remote.
- It should either extend java.rmi.server.UnicastRemoteObject class, or explicitly export itself.

The server application is used to create and make the remote object accessible to clients by registering it with local RMI registry. The class WFTP_ORBWorkManagerImpl is the client to RMIORBWorkManagerImpl. It obtains a reference to the remote server objects by looking up the RMI registry, then delegates the method call that it receives to this reference object. All other components' implementations follow this pattern.

2.5 METEOR Process Definition

The latest METEOR specification — Model 3 [Koc98] has incorporated many latest developments in workflow technology. In METEOR Model 3, a workflow design consists of three sub-designs:

1. Data: data include components of workflow instance data shipped among tasks, including portions of the overall data and some additional information, e.g., used for scheduling purposes.

2. Tasks (with or without associated maps): a task can be regarded as a unit of work, which is performed by a variety of processing entities, such as a computer program, a database transaction, or a network of interconnected tasks called a *workflow*, depending on the nature of the task.

3. Exception: an exception represents an occurrence of some abnormal event, and that can be detected by the underlying workflow management system. An exception is system-specific, known as system exception, if it may occur during the execution of every workflow, or is application-specific, known as application exception, if it is restricted to specific workflow applications.

An analogy exists between a programming language procedure and a workflow task. Like procedures, workflow tasks may have input/output parameters, used to transfer data in/out of the workflow task, or possibly the whole workflow. Like procedure call, a task may be invoked. A task invocation creates a task instance, involving specifying required and/or optional parameters. A task with multiple invocations is analogous to an overloaded procedures.

2.5.1 Task Realizations

A task may be realized by a computer program, by a human, or by workflow (network) of other tasks. Task realizations form a hierarchy, as shown in Figure 2.6. A task may be realized as *non-transactional*, *transactional*, or *workflow*. A *human* realization is a special case of a non-transactional realization, which is performed by a person. A composite realization is a special case of a workflow realization in which all of the human tasks are performed from the same terminal screen (most likely by the same person). An Open 2PC realization is a special case of a transactional realization where the participating transactional tasks follow the Open 2PC protocol.

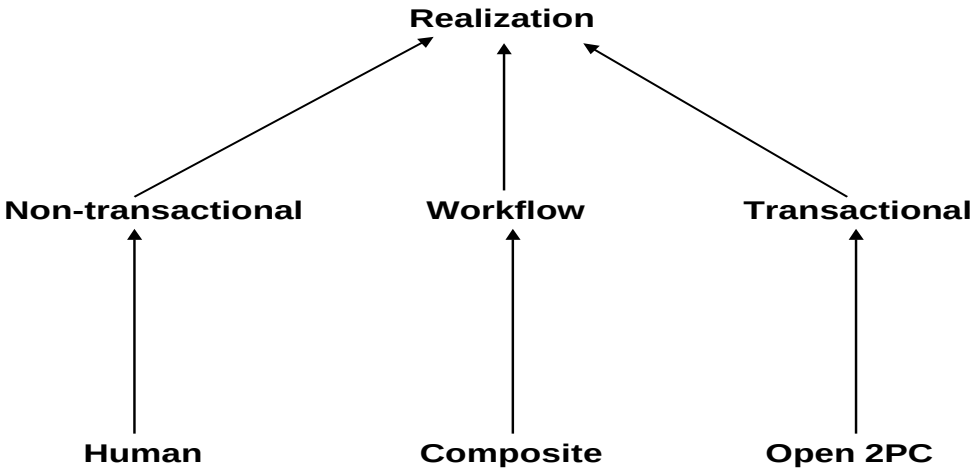

Figure 2.6: Task Realization Hierarchy

2.5.2 Designing Workflow Networks

A network represents the core of the workflow activity specification. It is a collection of tasks and transitions representing a sub-workflow. A *transition* joins two tasks and represents a transfer of activity in the workflow from the *source task* to the *destination task*. Precisely, the transition joins one of the final states in the source task with the initial state of the destination task.

A task may play a role of the source task in any number of transitions. Similarly, a task may be a destination task for a number of transitions. All of the transitions for which a task is the destination task are called the *input transitions* for that task. Likewise, all of the transitions for which a task is the source task are called its *output* transitions. All of the output transitions are further subdivided into *success*

transitions (if the task enters its Done or Commit state) and *failure transitions* (if the task enters its Fail or Abort state).

2.5.3 Routing

A transition may have an associated Boolean condition. The condition serves as a guard of the transition, i.e. the transition may be activated only if the condition is true. A *path* from task T_i to task T_j is a list of transitions leading from T_i to T_j, possibly via a number of intermediate tasks. A *cycle* is a path from task T to itself.

A group of input transitions is called *AND-join* if all of the participating transitions must be activated for the task to be *enabled* for realization. An AND-join is called *enabled* if all of its transitions have been activated. The number of transitions in an AND-join is called its *fan-in* number. An AND-join with a fan-in of N is said to *absorb* N parallel paths.

A group of transitions is said to have a *common source* if they have the same source task and all lead either from its success state, or its fail state with the same thrown exception. A group of common source transitions may form AND-split, OR-split, loop, or fork.

A group of common source transitions is called an *AND-split* if each of the transitions in the group has the condition set to true. It means that all of the transitions in the group are activated, once the task completes.

A group of common source transitions is called an *OR-split* (selection) is an ordered list of transitions where all but the last transition may have arbitrary conditions. The last transition on the list has the condition set to true. The selection of the transition to activate is done in the following way. If the condition of the first transition on the list evaluates to true, it is activated. If not, the second transition's condition is evaluated, and so on. Finally, if all of the conditions evaluate to false,

the last transition will be used, since its condition is always true. Transitions are attempted in the same order as they appear on the list.

A *loop* is a special case of an *OR-split*, where the list is composed of exactly two transitions. Structural constraint of a loop requires that *all* of the paths beginning with one of the transitions and *none* of the paths beginning with the other transition cycle back to the source task. Note that if a loop has both transitions set to true, the second transition is useless. Such a loop is infinite.

A *fork* is an ordered list of common source transitions in which the first transition has the condition set to *true*, and the remaining transitions may have arbitrary conditions. The semantics of a fork is as follows. Once the source task completes, *every* transition in fork for which the condition evaluates to true is activated. Transitions are attempted in the same order as they appear on the list.

2.5.4 Exception Handlers

An exception handler is a description of action(s) that the workflow enactment system, or possibly a workflow application, is going to perform in order to respond the exception. The most typical form of an exception handler is a *failure transition* — a transition in a network that leads from a failure state of some task.

It is possible that an abnormal event cannot be detected by any of the components of the workflow system. For example, a currently running workflow instance is in violation of a newly introduced business policy. It might be the case that the instance should fail the next task and continue its execution following one of its alternate paths. However, at this time neither the workflow application nor the workflow runtime system is not aware of the abnormal event. Such an abnormal even is called an external fault. An exception signal may be sent to the workflow system in order to make an external fault known to the workflow (either the runtime or the application). The signal may be sent by an external entity, such as a workflow administrator, or

a separate program. The workflow administrator may decide to force a particular workflow instance to fail one of its currently running tasks, and send a suitable exception to the task manager. Similarly, a process monitoring database updates may trigger sending an exception signal to the workflow manager. The workflow system receives the exception signal and continues its operation as if the exception was detected by the workflow itself.

2.6 WSFL

With the development of the Internet business, many business services are now available through the Web as Web Services. The Web Services Flow Language (WSFL) [Ley01] has been proposed by IBM to take advantage of existing web services to build inter-enterprise workflow systems. WSFL is an XML language for the description of Web Services compositions as part of a business process definition. It considers two types of Web Services compositions:

- The first type specifies the appropriate *usage pattern* of a collection of Web Services, and the result is typically a description of an executable business process known as a Flow Model. This model can be used to build intra-enterprise business systems.

- The second type specifies the *interaction pattern* of a collection of Web Services, and the result is a description of the overall *partner interactions* known as a Global Model. This model is used to build inter-enterprise workflow systems.

Full description of WSFL can be found in [Ley01].

WSFL is similar to METEOR process specification, and could be used as *external* process representation for METEOR if needed.

2.7 Exception Handling

Errors are a natural occurrence in any software system and WFMSs are no exceptions. The cause of errors in WFMSs are various, such as server crashing, task scheduler crashing, and network connection failures. In this chapter, we use `exception`, `fault` and `error` interchangeably. The usual failure-handling strategy in most systems is to stop execution and report the failure to the administrator. However, in large-scale complex workflow systems, this is not feasible due to its high demand of human resource and slow response time.

One attempt to achieve fault tolerance is to apply database transaction concept to workflow systems. Wheater et al. implemented the workflow execution environment as a transitional workflow system that enables sets of inter-related tasks to be carried out and supervised in a dependable manner [WSR98]. Their system servers as an example of the use of middleware technologies to provide a fault-tolerant execution environment for long running distributed applications. Muhlberger et al. found that backward recovery from the classical and advanced transaction management domains is applicable to workflow management, and that providing business level backward recovery is possible in the workflow domain [MOK99]. The transactional concepts are also used for error detection and recovery [DKM96] in the METEOR project [KS95] of our lab discussed in the next chapter. In METEOR, workflow exceptions are classified as infrastructure errors, workflow system errors, application and user errors. Unlike most WFMSs that treat task errors as internal within that task and cannot react based on the type of error that might have been returned by a task (e.g., errors resulting from incorrect input formats, logical errors at the task level, etc.), in METEOR, task errors are modeled and specified at the WFMS level. The error handling mechanism is based on a hierarchical error model. This makes it possible to *detect* and *handle* critical errors on the per-error basis. Therefore, errors are detected

and masked as close to the point of occurrence to prevent their propagation to other components. With this error model, task errors can be predefined by designers at build-time and handled at run-time [Wor97].

A recent approach aims at combining database transaction concepts and programming-language ideas into a coherent paradigm for fault-tolerant workflows [HA98]. This concept is based on enhancing the workflow model with new concepts such as atomic sphere in advanced transaction model, exception signals and exception handlers in programming languages [Alo00]. Atomic spheres are sets of activities that must either execute completely or not at all. It allows workflow designers to model processes as nestings of spheres, each of which can be separately undone if an error occurs during its execution, thus effectively limiting a failure's impact to the scope of that sphere. The concept of exception signals and handlers are borrowed from general programming languages like C++ and Java. It makes the workflow model more robust and comprehensible by separating normal control flow from the failure-handling logic. When an exception occurs in a sphere, the WFMS passes control to the appropriate handler, which executes the tasks necessary to repair the failure — it could abort the sphere, resume execution, or propagate the exception to the enclosing sphere [Alo00]. This approach is very similar to METEOR Model 3 discussed in section 2.5.

Defeasible workflow has also been proposed as a framework to support exception handling [LSK00], which gives directions to build exception-aware workflow systems. Defeasible workflow uses context dependent reasoning, together with a case-based-reasoning (CBR) mechanism with integrated human involvement, to enhance the exception handling capability of workflow management systems. This approach involves reusing the experience captured in prior exception handling cases.

2.8 Fault Tolerance

Although many research have been done in the area of exception handling, the redundancy issue has not been addressed adequately. As in other distributed systems, an important approach to handle exceptions is replication/redundancy. There are some faults that cannot be handled unless the system is redundant. For example, in case of server crash, the exception handler on that server is no longer available. For those critical components, such as a frequently accessed database, redundancy is indispensable to achieve reliability, otherwise, a single point of failure from these components could bring the whole system down. Unfortunately, this is a common characteristic of current WFMSs. Most systems lack the redundancy necessary to replace failed components without interrupting ongoing work. This thesis will exploit fault tolerance using replication, provide an implementation, and show how the overall availability of the workflow system is enhanced. In particular, this thesis will contribute in the following aspects.

- Present a framework for supplying fault tolerance to a workflow system.
- Develop a replication package that is reusable for general replication purposes.
- Implement fault tolerance as an enhancement to METEOR/ORBWork.

ORBWork uses a fully distributed scheduler and thus, introducing fault tolerance to this system requires investigation of fault tolerance techniques in distributed systems.

Chapter 3

Fault Tolerance in Distributed Systems

3.1 Introduction

Modern workflow products have evolved into complex distributed systems to achieve reliability, scalability, and support for heterogeneity. In this chapter, we will present a brief overview of general distributed systems, and then investigate extensively fault tolerance in distributed systems, as the necessary background to the following chapters.

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages [CDK01]. Distributed systems are everywhere nowadays. A group of computers that share a high quality laser printer in a lab form a distributed system, where each computer can communicate with the printer to print files. Actually, sharing resource such as printers or files is a big motivation for constructing distributed systems. The implementation of the Google search engine is another good example of a distributed system, where many component search engines cooperate to perform searches and later put their results together. Distributed systems are used to accommodate heterogeneity, provide scalability, achieve fault-tolerance and high availability. A distributed system should allow heterogeneous hardware and software coexist; if necessary, the system should be able to expand or shrink in number of components; and its services should be correct and highly available.

Our definition of distributed systems leads to the following characteristics of distributed systems: concurrency of components, and independent failures of components [CDK01]. Concurrency is an intrinsic property of distributed systems. Concurrency control is also an important issue when more than one process tries to update the same data at the same time. Independent failures result from the independence of components. One component may crash while others continue to function properly. We will take a closer look at failures in distributed systems, next.

3.2 Failures in distributed systems

All computer systems can fail, and we can do little to prevent failures from occurring entirely. System designers should plan for the consequences and appropriate handling of possible failures. Distributed systems can fail in many ways. Individual component failures, poor network connections, disconnected networks, etc., all contribute to system failures. Faults in the network result in the isolation of the computers that are connected to it, but that does not mean that they stop running. In fact the programs on them may not be able to detect whether the network has failed or has becomes unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a crash) is not immediately apparent to the other components with which it communicates. Therefore, the handling of failures is particularly difficult in distributed systems, in that the failures in a distributed system are most commonly partial –that is, some components fail while others continue to function [CDK01]. For example, in the sharing printer example, the printer may fail while all computers works properly. A user may not be able to visit the website outside a LAN due to the crash of the gateway that connects the LAN with the Internet, although he may still visit any website inside the LAN.

3.3 Providing Fault-Tolerance in Distributed Systems

Any process, computer or network may fail independently. Therefore each component needs to be aware of the possible ways in which its related components may fail and be designed to deal with each of these failures appropriately [CDK01]. Some failures are detectable, for example, checksums can be used to detect corrupted data in a messages or a file. In most cases, however, failures are not so obvious, and appropriate actions must be taken to handle them. Commonly used techniques for dealing with failures are:

- *Masking failures:* some failures can be masked and made less severe. Consider the following two examples:

  1. Email messages are re-sent when the deliveries fail.
  2. A web browser may simply ignore some unknown or wrong commands in HTML files.

- *recovery from failures:* Recovery involves the design of software so that the state of permanent data updated can be recovered or 'rolled back' after a server had crashed. In general, the computation performed by some programs will be incomplete when a fault occurs, and the permanent data that they update (files and other material stored in permanent storage) may not be in a consistent state [CDK01]. As an example, in scientific computation, long running programs usually write checkpoints periodically so that they can start from the last checkpoint before crash when they are restarted. This technique is also commonly used in database management where transactions should be either completely done or not done at all.

- *Tolerating Failures Using Redundancy:* services can tolerate failures if components are replicated and a sufficient right algorithm is in place. Consider the following examples:

    1. Yahoo! email services are replicated at multiple servers so that any single server failure will not bring down the whole system.
    2. In the Domain Name System, every name table is replicated in at least two different servers.
    3. A lab may have more than one printer accessible so that users may try others if the default one crashes.

    However, replication also introduces new challenges to the distributed system design. Replicas need synchronization to stay within a certain degree of consistency, which requires frequent communication for updating replica status and detecting peer failures, which in turn increases network overhead and response time. In case of replicating rapidly changing data, the overhead will decrease the performance a lot just to keep each replica up-to-date. Fault tolerance usually comes with high availability, since it involves providing services in spite of failures. Replication is an effective way to provide high availability.

Replication is a key to providing high availability and fault tolerance in distributed systems. High availability is of increasing interest with the tendency towards mobile computing and, consequently, disconnected operations. Fault tolerance is an abiding concern for services provided in safety-critical and other important systems [CDK01]. The rest of this chapter will concentrate on fault tolerance through replication.

3.4 Fault Tolerance Through Replication

We will study the replication of data: the maintenance of copies of data at multiple computers. Replication is a technique for enhancing services. It is a key to the effectiveness of distributed systems in that it can provide enhanced performance and fault tolerance [CDK01].

- *Performance enhancement:* The replication of Yahoo! email servers reduces the workload of each individual server by letting each replica server handle a subset of user requests, and, as a result, user waiting times for response are reduced. We cannot imagine that a single email server could handle millions of requests per second and still have a reasonable response time. Replication of immutable data is trivial: it increases performance with little cost to the system. Replication of changing data, such as that of the Web, incurs overheads in the form of protocols designed to ensure that clients receive up-to-date data. Thus, there are limits to the effectiveness of replication as a performance-enhancement technique [CDK01].

- *Fault tolerance:* Fault tolerance includes two facets:

    1. The first facet is *availability despite failures*. Also known as high availability. Services should be highly available. That is, the proportion of time for which the service is accessible with reasonable response times should be close to 100%. Replication is a technique for automatically maintaining the availability of data despite server failures. If data are replicated at two or more failure-independent servers, then client software may be able to access data at an alternative server, should the default server fail or become unreachable. That is, the percentage of time during which the service is available can be enhanced by replicating server data. If each of n

servers has an independent probability p of failing or becoming unreachable, then the availability of an object stored at each of these servers is:

$$1 - probability(\texttt{all servers failed or unreachable}) = 1 - p^n$$

For example, if there is a 10% probability of any individual server failure over a given time period and if there are two servers, then the availability is

$$1 - 0.10^2 = 1 - 0.01 = 99\%$$

which is a big improvement from the individual availability of 90%.

2. The other facet is providing *correct results despite certain misbehaviors*. Providing high availability of data may imply decreasing of the correctness of the data. The data may be out of date, as for example, an airline agent working offline may make reservations for seats that are already occupied and thus, the reservations must be reconciled when the agent gets back to online. A fault-tolerant service, by contrast, always guarantees strictly correct behavior despite a certain number and type of faults. The same basic techniques used for high availability – of replicating data and functionality across computer hosts – are also applicable to achieve fault tolerance. If up to N of $N+1$ servers crash, then in principle at least one remains to supply the service. And if up to N servers exhibit arbitrary or Byzantine failures – providing incorrect answers, even on purpose, then in principle a group of $2N + 1$ servers can provide a correct service, by having the correct servers outvote the failed servers (who may supplies spurious values) [LSP82]. But fault tolerance is subtler than this simple description makes it seem. The system must manage the coordination of its components precisely to maintain the correctness guarantees in the

face of failures, which may occur at any time. In this thesis, we should assume that no byzantine failures occur.

A common requirement when data are replicated is for *replication transparency*. That is, from the clients' point of view, there should be only a single copy of logical data, despite the fact that there actually exist multiple physical copies of data. Clients only need to identify one item when they request an operation to be performed. And clients also expect operations to return only one set of values [CDK01].

The other general requirement for replicated data – one that can vary in strength between applications – is that of consistency. This concerns whether the operations performed upon a collection of replicated objects produce results that meet the specification of correctness for those objects [CDK01]. Section 3.5 presents a general architecture for managing replicated data and it introduces group communication as an important tool. Group communication is particularly useful for achieving fault tolerance, which is the subject of Section 3.6.

3.5 System model and group communication

The data in our system consist of a collection of items that we call shared objects. An 'object' could be a file, a Java object, or some other data. Each such logical object is implemented by a collection of physical copies, called replicas. The replicas are physical objects, each stored at a single computer, with data and behavior that are tied to some degree of consistency by the system's operation. The 'replicas' of a given object are not necessarily identical, at least not at any particular point in time. Some replicas may have received updates that others have not received. In this section, we provide a general system model for managing replicas. In this model, replicas are viewed as a group representing a single logical object and each replica

plays a role in the group. Then we describe group communication systems, which are particularly useful for achieving fault tolerance through replication.

3.5.1 System Model

We make the following assumptions about the system:

1. The system is asynchronous, and processes may fail only by crashing, i.e., there are no byzantine failures.

2. Operations that a replica manager applies to its replica are recoverable, and deterministic. This allows us to assume that an operation at a replica manager does not leave any inconsistent results if it fails part-way through, and every replica reaches the same state after the same sequence of operations.

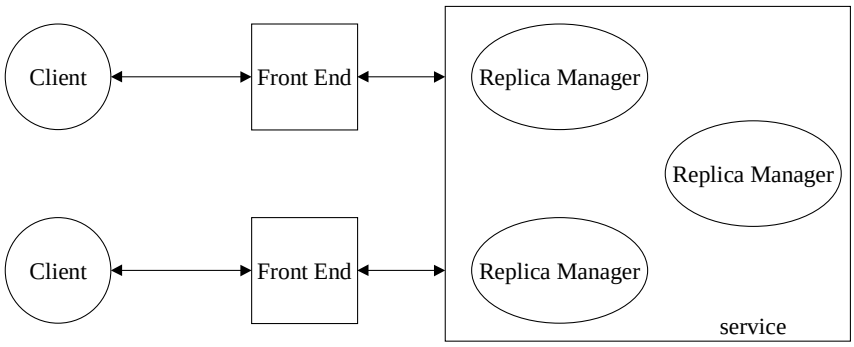

Figure 3.1: The General Architecture of Replication Model

We describe architectural components by their roles and do not mean to imply that they are necessarily implemented by distinct processes (or hardware). The model involves replicas held by distinct replica managers (see Figure 3.1), which

are components that contain replicas on a given computer and perform operations upon them directly.

The architecture is open in that replica managers may join or leave the system at any time. This is a imitation of real system where replica managers may crash and leave the system, and they may be restarted and join the system again.

The general model of replica management is shown in Figure 3.1. A collection of replica managers provide service to clients. The clients see a service that gives them access to objects (for example, diaries or bank account), which in fact are replicated at the managers. Each client requests a series of operations – invocations upon one or more of the objects. An operation involves a combination of reads of objects and updates to objects. Requested operations that involve no updates are called read-only requests; requested operations that update an object are called *update operations* (these may also involve reads) [CDK01].

Requests from clients are first handled by a component called the *front end*. The role of the front end is to communicate by message passing with one or more of the replica managers, rather than forcing the client to do this itself explicitly. It is the vehicle for making replication transparent [CDK01]. The front end may be implemented on the client side or on the server side.

In general, five phases are involved in the performance of a single request upon the replicated objects [WPS00]. Depending on the specific system, the actions in each phase vary. For example, a service that provides high availability behaves differently from one that provides a byzantine-fault-tolerant service. The phases are as follows:

- *The front end issues the request to one or more replica managers.* The front end can communicate with a single replica manager, which in turn communicates with other replica managers, or it can multicast the request to all the replica managers.

- *Coordination:* The replica managers coordinate in preparation for executing the request consistently. They agree, if necessary at this stage, on whether the request is to be applied (it might not be applied at all if a failure occurs at this stage or it is a duplicated request that has been performed). They also decide on the ordering of this request relative to others. Possible orders are as follows:

  - *FIFO ordering:* if the front end issues request r then request r', then any correct replica manager that handles r' handles r before it.
  - *Causal ordering:* if the issuing of request r happened-before the issuing of request r', then any correct replica manager that handles r' handles r before it.
  - *Total ordering:* if a correct replica manager handles r before request r', then any correct replica manager that handles r' handles r before it.

  Most applications require FIFO ordering.

- *Execution:* The replica managers execute the request, ideally, in such a way that they can undo its effects later if failures occur.

- *Agreement:* The replica managers reach consensus on the effect of the request – if any – that will be committed. For example, in a transactional system, the replica managers may collectively agree to abort or commit the transaction at this stage.

- *Response:* One or more replica managers respond to the front end. In some systems, one replica manager sends the response. In others, the front end receives responses from a collection of replica managers and it selects or synthesizes a single response to pass back to the client. For example, it could pass back the

first response to arrive, if high availability is the goal. If tolerance of byzantine failures is the goal, then the front end could give the client the response that a majority of the replica managers provides [CDK01].

3.5.2 Group Communication

Group communication is a powerful concept for managing replicated data. It is an application of Object-Oriented Paradigm: replicas are encapsulated within a group that provides an interface to manipulate these replicas. In a group that manages replicated data, for example, users may add or withdraw a replica manager, or a replica manager may crash and thus, need to be withdrawn from the system's operation. A full implementation of group communication incorporates a *group membership service* to manage the dynamic membership of groups [CDK01]. Three important tasks of group membership services are as follows:

- *Providing an interface for group membership changes:* the membership service provides operations to create and destroy process groups and to add or withdraw a process to or from a group. In most systems, a single process may belong to several groups at the same time.

- *Implementing a failure detector:* the service incorporates a failure detector. The service monitors the group members not only in case they should crash but also in case they should become unreachable because of a communication failure. The detector marks processes as Suspected or Unsuspected. The service uses the failure detector to reach a decision about the group's membership: it excludes a process from membership if it is suspected to have failed or to have become unreachable.

- *Notifying members of group membership changes:* the service notifies the group's members when a process is added, or when a process is excluded.

Systems that can adapt without service interruption as processes join, leave and crash – fault-tolerant systems, in particular – require the more advanced feature detection and notification of membership changes. A full group membership service maintains *group views*, which are lists of the current group members, identified by their unique process identifiers. A new group view is generated when processes are added or excluded.

It is important that a group membership service may exclude a process from a group because it is suspected, even though it may not have crashed. A communication failure may have made the process unreachable, while it continues to execute normally. A membership service is always free to exclude such a process. A false suspicion of a process and the consequent exclusion of the process from the group may reduce the group's effectiveness. The group has to manage without the extra reliability or performance that the withdrawn process could have potentially provided [CDK01].

Although this is not its main goal, the Sun Microsystems' Jini system [Jini] provides a good facility to implement group communication, fulfilling all three requirements above:

1. *Leased registration interface:* with this interface, replica managers register join a group by registering their replicas with a registry for a certain period of time, and renew the registration if it wants to stay in the group. By cancelling or not renewing its registration, a replica manager leaves the group.

2. *Failure Detection:* if a replica manager fails to renew its registration, it is assumed to fail and is removed from the registry, thus excluded from the group.

3. *Notifying Membership Changes:* if a replica manager registers its interest in membership changes, it will be notified when a process is added or excluded.

3.6 FAULT-TOLERANT SERVICES

In this section, we examine how to provide a service that is correct despite up to N process failures, by replicating data and functionality at replica managers. For the sake of simplicity, we assume that communication remains reliable and that no network partitions occur. Each replica manager is assumed to behave according to a specification of the semantics of the objects it manages, when they have not crashed. In other words, no byzantine failures occur.

3.6.1 PASSIVE (PRIMARY-BACKUP) REPLICATION

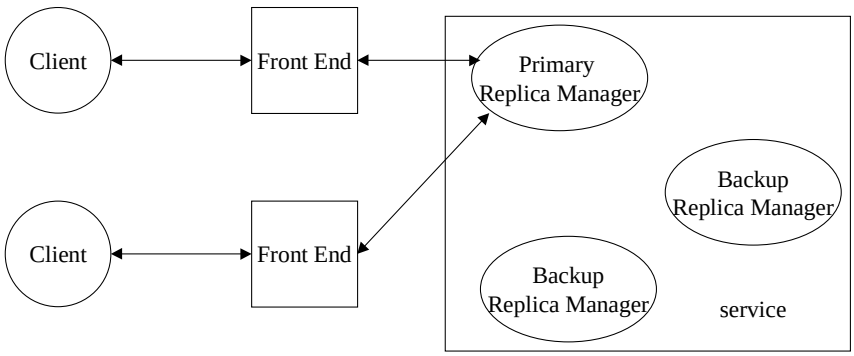

Figure 3.2: Passive (primary-backup) Replication

In the *passive* or *primary-backup* model of replication for fault tolerance, at any one time there is a single primary replica manager and one or more secondary replica managers – 'backups' or 'slaves'. In the pure form of the model, front ends communicate only with the primary replica manager to obtain the service, as shown in Figure 3.2. The primary replica manager executes the operations and sends copies

of the updated data to the backups. If the primary fails, one of the backups is promoted to act as the primary [CDK01].

The sequence of events when a client requests an operation to be performed is as follows:

1. *Request:* the front end issues the request, containing a unique identifier, to the primary replica manager.

2. *Coordination:* the primary takes each atomic request, in the order in which it was received. It checks the unique identifier, in case it has already executed the request and if so it simply re-sends the response.

3. *Execution:* the primary executes the request, logs the updates if any, and stores the response.

4. *Agreement:* if the request is an update then the primary sends the updated state, the response and the unique identifier to all the backups. Each backup sends an acknowledgement.

5. *Response:* the primary responds to the front end, which sends the response back to the client.

This system obviously implements linearizability if the primary is correct, since the primary sequences all the operations upon the shared objects. If the primary fails, then the system retains the linearizability if a single backup becomes the new primary and if the new system configuration takes over exactly where the last left off [CDK01]:

- the primary is replaced by a unique backup (if two clients began using two backups, then the system could perform incorrectly); and

- the replica managers that survive agree on which operations had been performed at the point when the replacement primary takes over.

Consider a front end that has not received a response. The front end retransmits the request to whichever backup takes over as the primary. The primary may have crashed at any point during the operation. If it crashed before the agreement state 4, then the surviving replica managers cannot have processed the request. If it crashed during the agreement stage, then they may have processed the request. If it crashed after that stage, then they have definitely processed it. But the new primary does not have to know what stage the old primary was in when it crashed since it has already received all the updates. When it receives a request, it proceeds from stage 2 above. No consultation with the backups is necessary, because they have all processed the same set of messages.

Discussion of passive replication

To survive up to N process crashes, a passive replication system requires $N + 1$ replica managers (such a system cannot tolerate byzantine failures). The front end requires little functionality to achieve fault tolerance. It needs to be able to look up the new primary when the current primary does not respond.

Passive replication has the disadvantage of suffering from a relatively large overhead because much effort has to be done to maintain a single valid primary replica. The primary replica must be closely monitored so that it can be replaced quickly in case of failures.

3.6.2 Active replication

In the *active* model of replication for fault tolerance, the replica managers are state machines that play equivalent roles and are organized as a group. Front ends multicast their requests to the group of replica managers and all the replica managers

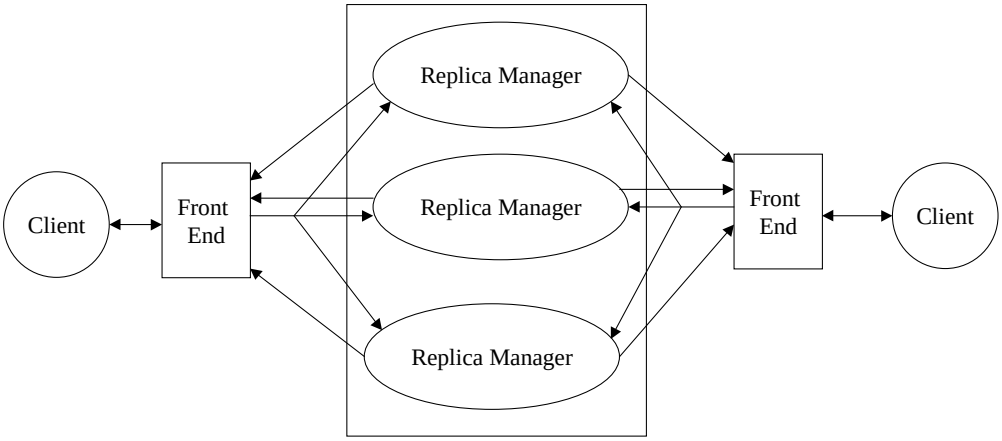

Figure 3.3: Active Replication

process the request independently but identically and reply, as shown in Figure 3.3. If any replica manager crashes, then there is no impact upon the performance of the service, since the remaining replica managers continue to respond in the normal way. We shall see that the active replication can tolerate byzantine failures, because the front end can collect and compare the replies it receives [CDK01].

Under active replication, the sequence of events when a client requests an operation to be performed is as follows:

1. *Request:* the front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. It is assumed that the worst failure of the front end is its crash. It does not issue next request until it has received a response.

2. *Coordination:* the group communication system delivers the request to every correct replica manager in the same (total) order.

3. *Execution:* every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.

4. *Agreement:* no agreement phase is needed, because of the multicast delivery semantics.

5. *Response:* each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and on the multicast algorithm. If, for example, the goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest (it can distinguish these from responses to other requests by examining the identifier in the response).

3.7 SUMMARY

In this chapter, we first investigated distributed systems and their fault tolerance in general. Then we focused on the replication technique in depth, and introduced the general model and two special models: passive and active replication models. We will see how these techniques are used to supply fault tolerance to our workflow system in the rest of the thesis.

Chapter 4

Introducing Fault Tolerance To ORBWork

In section 3.4 we have investigated in depth the replication techniques in distributed systems. Also, we have discussed Workflow Management Systems(WFMSs) and, the ORBWork system in particular. In this chapter, we are going to combine all these together by applying the replication models to the ORBWork system to achieve fault tolerance. We will begin with an introduction to Jini, the core implementation technology we used.

4.1 Introduction to Jini

Jini is the name for a distributed computing environment that can offer "network plug and play". In a Jini system, there are three main components: a *service*, such as a printer, a *client*, which would like to make use of this service, and a *lookup registry* (service locator) which acts as a locator between services and clients [New00]. Obviously, they have to be connected by a network so they can interact with each other. The service can announce its presence by registering with the lookup. And the client can locate the service by looking up the service name in the lookup registry. The Jini architecture suits for any network where some degree of change is desired.

Jini is distinguished by being based on Java, and usually implemented on top of Java RMI, although this is not specified in Jini architecture. Unlike Java RMI, Jini lookup registry accepts registrations from all over the network. It also supports more advanced searches. Clients can use templates to locate the desired services. The

search result could be a single service, or a set of services, depending the specification in the template supplied by the client. This is especially useful for replication, where all replicas register with the same group name, but they may also specify different values for some attributes, such as whether or not they are the primary replica. Like in Java RMI, proxy code will be moved around between the three players.

4.1.1 Service Registration

The server needs to register the service with the lookup service. It must first find a lookup service. This can be done in two ways: if the location of the lookup service is known, then the server can use Unicast TCP to connect directly to it. If the location is unknown, the server will make UDP multicast requests to the default port 4160, and lookup services will respond to these requests by sending back to the server a proxy object known as a registrar. The server then uses the registrar to upload/register all the information necessary for the service, including the service name and its proxy object.

A server application will internally perform the following steps [New00]

```
prepare for discovery
discover a lookup service
create information about a service
export a service
renew leasing periodically
```

4.1.2 Client Lookup

The client goes through the same mechanism to get a registrar from the lookup service. Then it looks up the service by the service name and gets a service proxy downloaded to it. With this service proxy, the clients then can make requests of the service object in its own JVM.

A client application will internally perform the following steps [New00]

```
prepare for discovery
discover a lookup service
prepare a template for lookup search
lookup a service
call the service
```

4.1.3 Support Services

As we have seen above, class definitions for service proxy objects must be downloadable from where the service came from. This is commonly done using an HTTP or FTP protocol. So, Jini requires an HTTP/FTP server running on the URL where the service class files are stored. The other support service is the RMI daemon. A proxy service gets exported to the client and communicates back to its host service. There are many ways to do this. One way is the Java RMI system. In order to run reggie, the lookup service supplied by Sun that exports registrar objects, you have to start an rmid server, then start reggie on the same machine, and reggie will register with the RMI daemon.

4.2 The Replica Package

Based on the same procedures of registering services and looking up services, a convenience package called replica has been developed to facilitate replication. This convenience package is generally applicable to any kind of replication, and is a contribution of this thesis. Table 4.1 lists all the classes in this package.

Based on the common characteristics between these two replication models, the interface PassiveReplicaIf is a subclass of ActiveReplicaIf, and PassiveReplicaImpl a subclass of ActiveReplicaImpl. For more information about the implementation of the replica package, see appendix 5.

ActiveReplicaIf	The interface for services that are actively replicated
ActiveReplicaImpl	A basic implementation for ActiveReplicaIf, Actively replicated services should extend this class
ActiveReplicaManager	The manager used to manage actively replicated services
ActiveReplicaLocator	The locator for actively replicated services
PassiveReplicaIf	The interface for services that are passively replicated
PassiveReplicaImpl	A basic implementation of PassiveReplicaIf, Passively replicated services should extend this class
PassiveReplicaManager	The manager used to manage passively replicated services
PassiveReplicaLocator	The locator for passively replicated services

Table 4.1: Convenience Classes in the Replica Package

4.3 Fault Tolerance in ORBWork

ORBWork consists of the monitor, ORBWork manager, ORBWork server, worklist server, data server, and a number of task schedulers and task managers. All of these components need to be replicated to provide fault-tolerance. In principle, any component can be replicated using either the active or passive replication model. Depending on the semantics, however, we decided to replicate the task schedulers, ORBWork manager and the monitor using passive replication, and the rest of the components using active replication. The reason is that we do not want to have more than one active ORBWork manager governing the system at the same time, although many backup ORBWork managers can exist besides the primary one. Not all instances should be executed replicatedly, so the task schedulers are replicated passively. The monitor is also replicated passively because there is only one log file specified in the property file, and it will incur inconsistency if all replica monitors are trying to update the same log file. The rest components are replicated actively because active replications incur less network overheads.

Recall the WFTP structure from section 2.2. Here, we will see how the replication is implemented within that structure.

For the passively replicated components, including the task schedulers, ORB-Work manager and monitor, the replication is done as follows.

- RMI interfaces must extend the replica.PassiveReplicaIf directly or indirectly.
- RMI implementations must extend the replica.PassiveReplicaImpl directly or indirectly.
- Server application should register the remote service object

  using replica.PassiveReplicaManager The relevant code snippet is shown below:

```
PassiveReplicaManager pmgr=new PassiveReplicaManager();
pmgr.registerObject("/ORBWork/Admin//Monitor.svc", serviceRef);
```

- WFTP implementations should use replica.PassiveReplicaLocator to locate the primary replica. The relevant code snippet is shown below:

```
PassiveReplicaLocator server = new PassiveReplicaLocator();
server.set_criterion("/ORBWork/Admin//Monitor.svc", null);
MonitorIf monitor = (MonitorIf)server.getRef();
try{
  monitor.RecordTaskStatus(WorkflowName,
                           WorkflowInstanceId,
                           TaskName,
                           HostName,
                           state);
}
.....
```

Similarly, for the actively replicated components, we have

- RMI interfaces must extend the replica.ActiveReplicaIf directly or indirectly.

- RMI implementations must extend the replica.ActiveReplicaImpl directly or indirectly.

- Server application should register the remote service object

  using replica.ActiveReplicaManager

```
ActiveReplicaManager amgr=new ActiveReplicaManager();
amgr.registerObject("/ORBWork/WorklistMgr.svc", serviceRef);
```

- WFTP implementations should use replica.ActiveReplicaLocator to locate the primary replica. The relevant code snippet is shown below:

```
ActiveReplicaLocator server = new ActiveReplicaLocator();
server.set_criterion("/ORBWork/WorklistMgr.svc", null);
Object[] wlserver=server.getAllRef();
for(int i=0;i<wlserver.length;i++){
  try{
    ((WorklistIf)wlserver[i]).PostItem(WorkflowId,
                                       WorkflowName,
                                       TaskName,
                                       Data);
  }catch(Exception e){
  }
}//for loop
```

Note, in the RMI version, all of the RMI components but the data server are subclasses of RMIORBWorkComponentImpl. Since Java does not allow multiple inheritance, we simply make RMIORBWorkComponentImpl a subclass of PassiveReplicaImpl. So, the passively replicated components like ORBWork managers inherit PassiveReplicaImpl indirectly. Since PassiveReplicaImpl is a subclass of ActiveReplicaImpl, the actively replicated components like ORBWork servers also inherit ActiveReplicaImpl indirectly. The class diagram is shown in Figure 4.1.

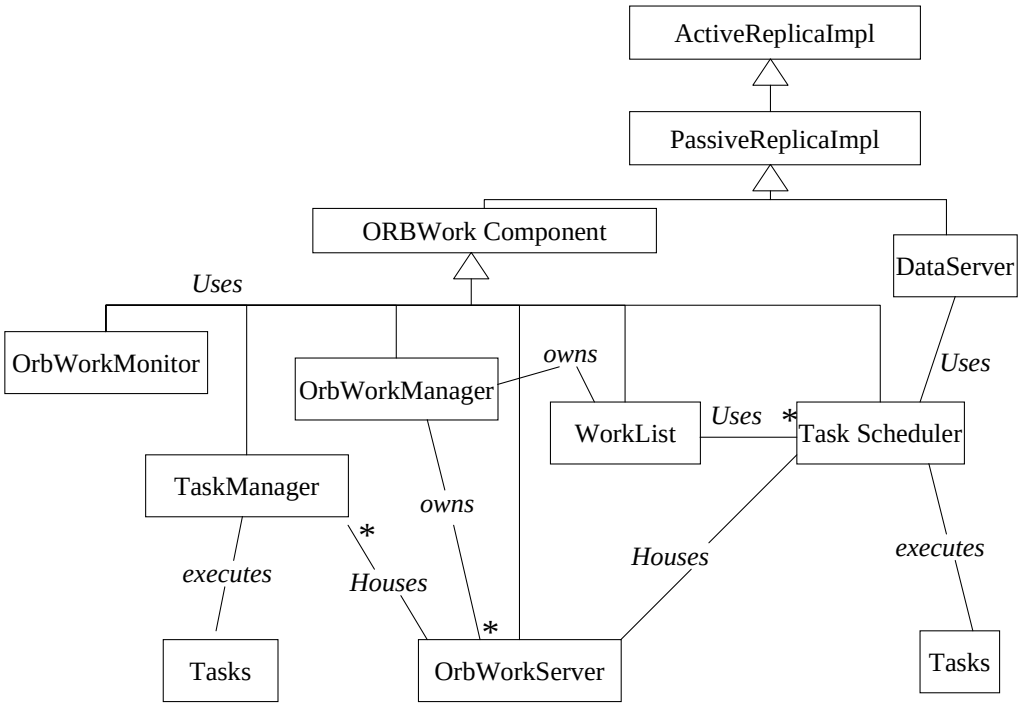

Figure 4.1: Class Diagram of Replicated ORBWork Components

4.4 Fault Tolerant Transitions

A transition involves two task schedulers, however, in case of replication, it involves two groups of task schedulers, e.g., a group of predecessor task schedulers and a group of successor task schedulers. Each group, and each replica play their own role during this transition. One of the replicas assumes the primary responsibility, sometimes called the primary replica; the backup replicas assume the backup responsibility. When a transition is finished, the responsibility is shifted to the successor group and the predecessor group is relieved of the responsibility. Many failures could happen

during this transition, and the following 4 steps are designed to deal with these failures.

1. The primary predecessor task scheduler requests a transition to the primary successor. Now, the responsibility is on the group of predecessor replicas. The primary predecessor assumes the primary responsibility.

2. The primary successor receives the transition input. It checks if this transition has happened before. If it has, it simply returns. The primary successor notifies its backups of the transition by passing them the transition input. It then processes the transition locally, as in non-fault-tolerant ORBWork, putting the transition into the local processing queue.

3. The primary successor sends back an acknowledgement to the primary predecessor. Now, the successor assumes the responsibility.

4. The primary predecessor then notifies its backups that the transition is done, and they can release the responsibility now.

These four steps exhibit resemblance with the five steps of processing a request in the passive replication model. In fact, this is actually an application of that model: the primary predecessor is the client of the successor task scheduler. If the predecessor does not receive a response from the primary successor, it tries again. To better understand these algorithms, we will take the SimpleFlow as an example. SimpleFlow is a small test workflow, containing three tasks, as well as a single parent task implementing the whole workflow. The three tasks are Start, Process, and Stop. In normal execution, an instance will go through the sequence of transitions, as shown in Figure 4.2. First, the workflow manager starts an instance by invoking the Transition method of SimpleFlow task scheduler. SimpleFlow then makes a transition (through a task manager) to Start, then Start to Process, Process to Stop, Stop back to

SimpleFlow, finally SimpleFlow notifies the workflow manager that the instance is processed. Figures 4.3 through Figure 4.8 show how the transitions are done between

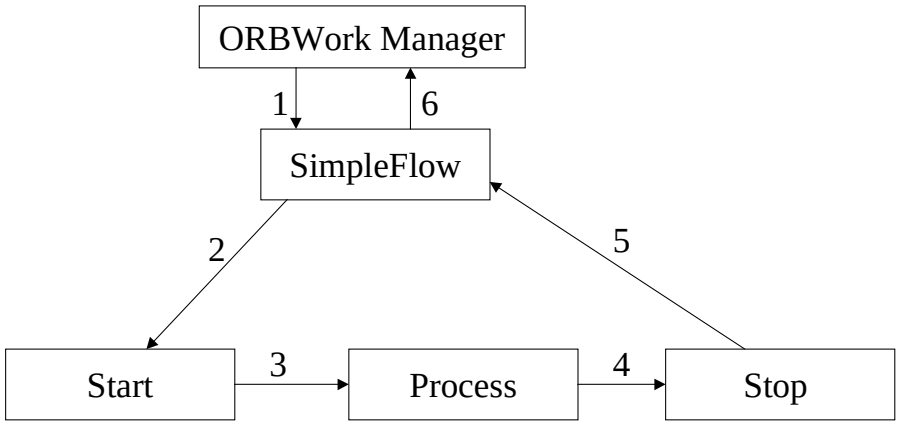

Figure 4.2: Normal execution sequence of SimpleFlow

tasks in normal execution.

1. *ORBWork Manager Coordination*: as shown in Figure 4.3, the user sends request to the primary ORBWork manager for a new instance. The primary ORBWork Managers then notifies all its backups that a transition is to be done. The responsibility is now on the group of managers.

2. *Transition between ORBWork manager and SimpleFlow*: as shown in Figure 4.4, the primary ORBWork manager makes a transition to the primary SimpleFlow which notifies the backup SimpleFlows of this transition and sends an acknowledgement back to the primary ORBWork manager. The primary ORBWork manager then notifies its backups that the transition is done, and they are relieved of the responsibility. Now the responsibility is shifted to the group of SimpleFlow task schedulers.

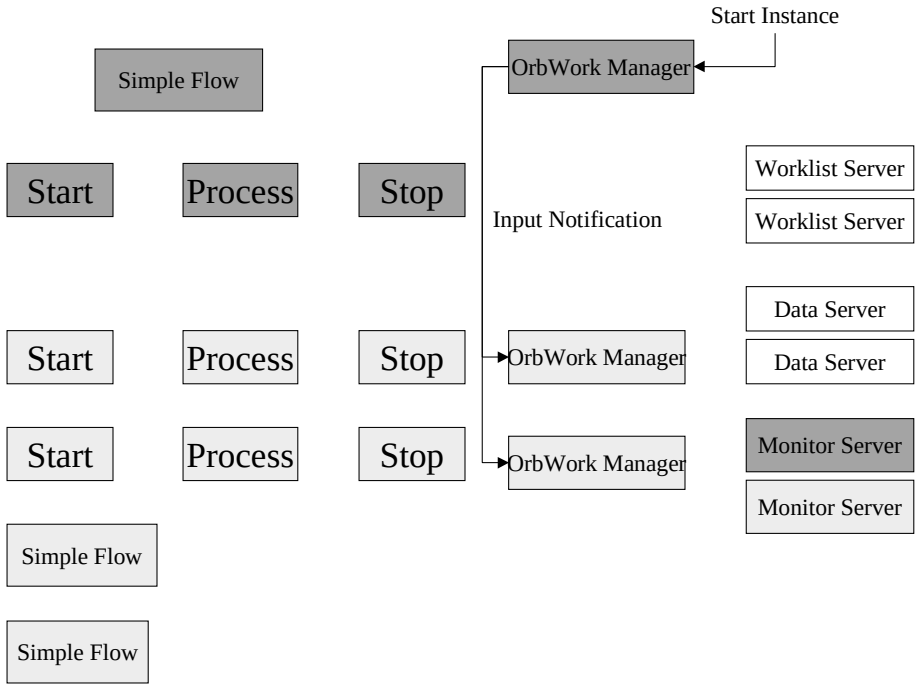

Figure 4.3: ORBWork Manager Coordination

3. *Transition between SimpleFlow and Start*: as shown in Figure 4.5, the primary SimpleFlow makes a transition to the primary Start which in turn notifies the backup Starts of this transition and sends an acknowledgement back to the primary SimpleFlow. The primary SimpleFlow then notifies its backups that the transition is done, and they are relieved of the responsibility. Now the responsibility is shifted to the group of Start task schedulers.

4. *Transition between Start and Process*: as shown in Figure 4.6, the primary Start makes a transition to the primary Process which in turn notifies the backup Processes of this transition and sends an acknowledgement back to the

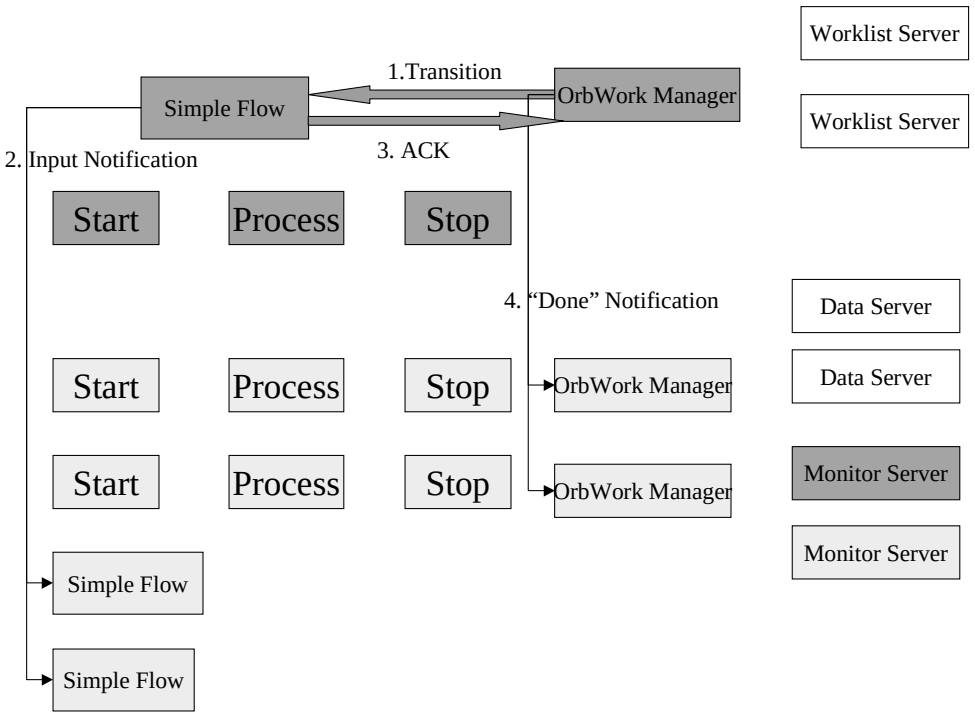

Figure 4.4: Transition between ORBWork manager and SimpleFlow

primary Start. The primary Start then notifies its backups that the transition is done, and they are relieved of the responsibility. Now the responsibility is shifted to the group of Process task schedulers.

5. *Transition between Process and Stop*: as shown in Figure 4.7, the primary Process makes a transition to the primary Stop which in turn notifies the backup Stops of this transition and sends an acknowledgement back to the primary Stop. The primary Process then notifies its backups that the transition is done, and they are relieved of the responsibility. Now the responsibility is shifted to the group of Stop task schedulers.

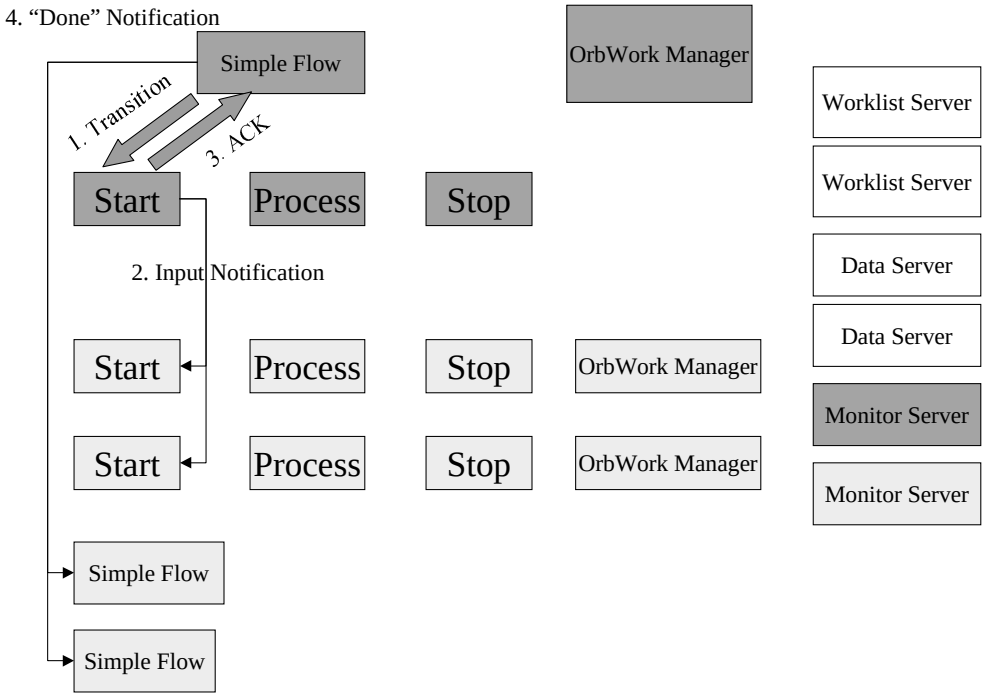

Figure 4.5: Transition between SimpleFlow and Start

6. *Transition between Stop and SimpleFlow*: as shown in Figure 4.8, the primary Stop makes a transition to the primary SimpleFlow saying that the instance is processed completely. The primary SimpleFlow in turn notifies the backup SimpleFlows of this transition and sends an acknowledgement back to the primary Stop. The primary Stop then notifies its backups that the transition is done, and they are relieved of the responsibility. Now the responsibility is shifted to the group of the SimpleFlow task schedulers. The primary SimpleFlow will report this message to the ORBWork manager.

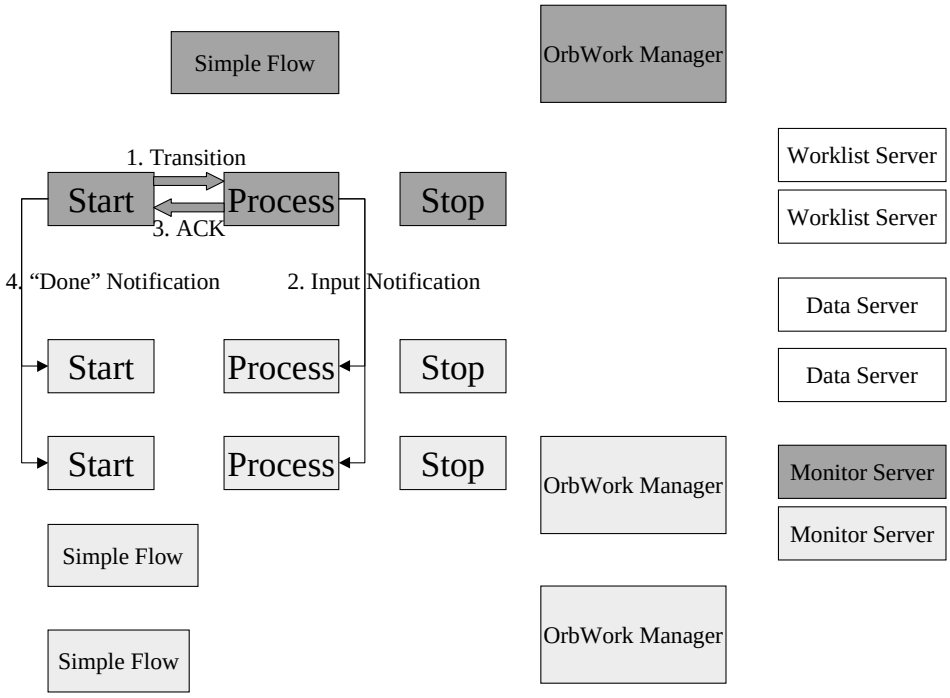

Figure 4.6: Transition between Start and Process

4.5 Failure handling

Now let's take a look at how these algorithms handle failures during a transition. Possible failures are listed below

1. The primary predecessor crashes before initiating the transition
2. The primary successor crashes before notifying its backups of the transition
3. The primary successor crashes while processing the transition
4. The primary predecessor crashes while waiting from the acknowledgement

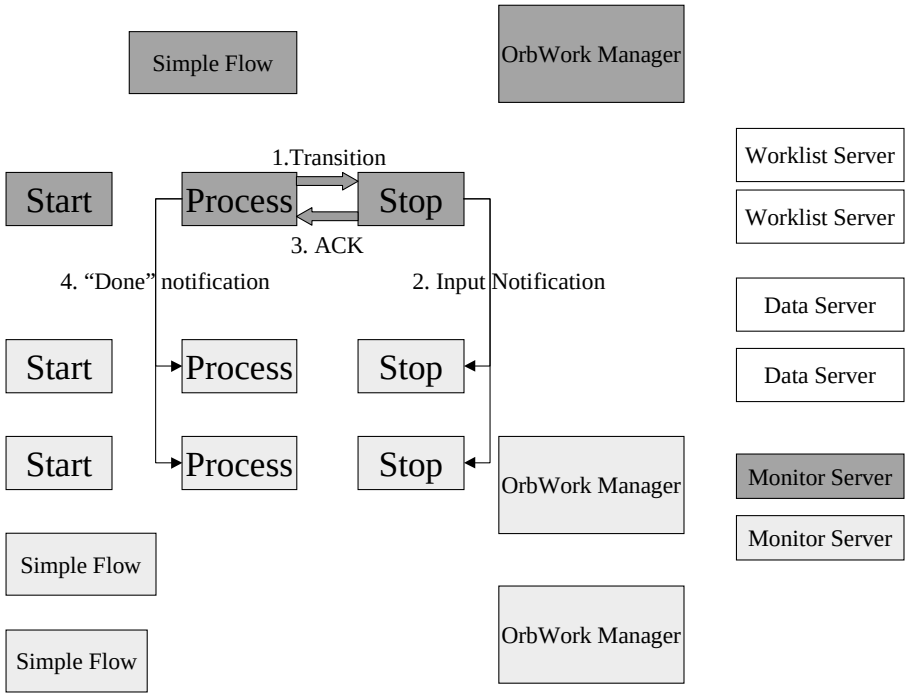

Figure 4.7: Transition between Process and Stop

5. The primary predecessor crashes before notifying backups of ”Done” message

6. Network partitions

4.5.1 Primary predecessor crashes before invoking primary successor

The replica managers of the predecessor group will establish a new primary predecessor which will recover from the failure point and eventually will come to the point of making the transition.

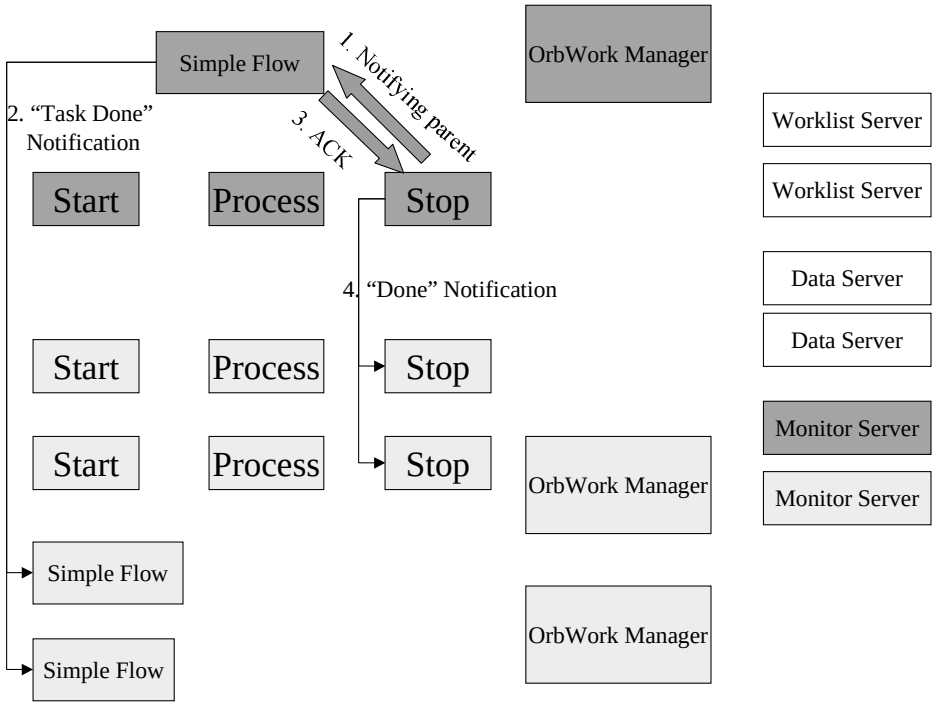

Figure 4.8: Transition between Stop and SimpleFlow

4.5.2 Primary successor crashes just before notifying its backups of the transition

Similarly, the replica managers of the successor group will establish a new primary successor. Meanwhile, an exception will be caught in the primary predecessor. The primary predecessor calls replica locator to find the reference to the new primary successor, and tries to make the transition again.

4.5.3 Primary successor crashes while processing the transition

Again, an exception will be caught in the primary predecessor. The primary predecessor calls replica locator to find the reference to the new primary successor, and makes a transition to it. Meanwhile, the newly established primary successor may already start processing the transition as a recovery step since it does not get the "done" message, or may be just invoked by the primary predecessor. Either way works. And the transition will not be duplicated since a duplication check is performed.

4.5.4 Primary predecessor crashes while waiting for the acknowledgement

The newly established primary predecessor will make the transition again. The primary successor will find out that the transition has already been done and simply return an acknowledgement.

4.5.5 Primary predecessor crashes just before notifying backups of "done" message

Like the previous case, The newly established primary predecessor will make the transition again. The primary successor will find out that the transition has already been done and simply return an acknowledgement.

Chapter 5

Implementing Passive Replication Model

In this chapter, we will look at the implementation of replication models in more detail. We will only talk about the implementation of the passive replication model because that of the active replication model is much simpler in terms of implementation. In the passive replication model, there are a number of replicas that can perform identical services. They form a replica group, and become peers to each other. Only one of them is designated as the primary at any time, while the rest are backups. The replication is transparent to clients. Clients can obtain a reference to the group and invoke the group to perform services. It is left to the group to decide how each replica responds to the invocation. The backups watch the primary replica—periodically pinging the primary to see if it is still up. Once the primary replica crashes, one of the backups is elected as the new primary replica and then continues from the failure point left by the original primary. Besides the replicas, there are also several (at least one) lookup registries in the model. All replicas register with the lookup registries using the same group name upon joining the system. The group is then identified by the group name. Each replica also has to renew its registration with registries periodically, otherwise its entry will expire and be removed from registries. Replicas can locate their peer replicas with the help of registries.

There are two approaches to implement this model. One is to have the replicas themselves do everything, including group member service (such as registration, communicating with other replicas, etc.) and performing services. The other is to

separate the group member service from performing services. A replica just performs services, and a replica manager manages the replica by handling group member service. A replica locator is also provided for clients to find a reference to the group. Clients only need to specify the group name. This thesis uses the second approach in the attempt to make as few changes to the original code as possible. The rest of this chapter will focus on the implementation algorithms.

5.1 Replica Object

Replica objects are the actual service providers. They are managed by replica managers. All service replica interfaces should extend the *PassiveReplicaIf* interface, which defines basic methods for replica managers to manage replica objects. All service implementations should also extend the *PassiveReplicaImpl* class, which provides the basic implementation of a replica. Below is a list of methods defined in PassiveReplicaIf.

1. `getID()`,return the registration ID
2. `setID(ID)`,used by replica manager to set the registration ID
3. `update(status)`,used by primary replica to notify status changes
4. `is_primary()`,return a boolean value to indicate whether it is primary replica
5. `set_primary(boolean)`,used by replica manager to set the replica as primary or backup
6. `register(backup)`,used by backups to register themselves with the primary
7. `getHost()`,return the host name, used to obtain the host name of the replica

The basic implementation of the interface in class `PassiveReplicaImpl` follows. There are two important fields:

1. myRole, true if the replica is primary, false if not, set by replica manager

2. myID, the registration ID of this replica

5.1.1 CHECK IF A REPLICA IS PRIMARY

This method is provided to check if a replica is primary.

```
ALGORITHM is_primary
  return myRole
```

5.1.2 SET THE ROLE OF A REPLICA

This method is provided to set the role of a replica. If the replica is set as primary, it should call the method `recover` as an attempt to correct the failure if any.

```
ALGORITHM is_primary(boolean role)
  myRole <- role
  if myRole is primary
    call recover
```

5.1.3 SET AND GET ID

Like the above two methods, these two methods are straightforward.

```
ALGORITHM setID(ID)
  myID <- ID


ALGORITHM getID()
  return myID
```

5.1.4 UPDATE STATUS AND RECOVERY

The primary replica notifies its backups when it starts or finishes a service. Before performing a service, or, at the beginning of a method invocation, the primary replica

packs the method name and all parameters into a list and passes this list to every backup by calling the method *update*. When the service is successfully done, the primary passes a "done" message to every backup. This is necessary for recovery purposes. As the backups know the failure point left by the original primary, the new primary knows from where to recover the failure. If the status says "done", then no recovery is necessary. The method *update* is fairly simple. Just record the status received from the primary.

```
ALGORITHM update(status)
  my_status <- status
```

The recover method should be re-implemented by subclasses since the recovery mechanism heavily depends on the nature of services.

5.2 Replica Manager

As mentioned above, the replica managers manage replicas, including registering them with registries, making sure there is only one primary replica in the group, etc. There are some important fields listed below.

```
impl : the replica managed by the replica manager
myID : the registration ID of impl
primary_ref : the reference to the primary replica
primary_id : the registration ID of the primary replica
```

5.2.1 Announce presence

Upon joining the system, a replica manager announces the presence of *impl* using multicast. The pseudo code follows.

```
ALGORITHM announce_presence
1  Prepare for discovery of lookup registries using multicast.
2  Add a listener to discovery event
```

```
3  Create information about impl, including group name,
     whether primary or backup, and possibly other information.
4  Register impl with every registry with the
     same registration ID
5  Record this lookup into a local vector of lookup references
6  Renew the registration periodically to
     maintain the entry in the lookup
```

If a replica manager fails to renew its registration, lookup registries will assume it has crashed and remove its entry. So, the information in the lookup is always close to up-to-date. Two issues need to be noticed. First, how does a replica get its registration ID? Initially the registration ID is *null*, after registration with the first registry, a 128-bit random service ID is assigned by the registry. And the replica manager will use the same ID for all future registration. Since the ID is such a long random string, we do not have to worry about ID conflicts. Second, how does a task scheduler know whether it should register as primary or backup? We simply let each replica firstly be registered as a backup, then if it finds exception as a result of pinging the primary, a search for a primary replica will be conducted, as described next.

5.2.2 Ping the primary replica

A replica manager periodically pings the primary replica, and takes appropriate action if the primary crashes. The pinging is accomplished by invoking the `is_primary` method. If the method is invocable, the primary is alive; otherwise, the primary crashes or becomes unaccessible. Every 5 periods, a verification is performed to verify that there is one and only one primary in the group. The pseudo code is given below.

```
ALGORITHM PING THREAD
1  loop for ever
2    sleep for a period //a period can be specified
```

```
 3   sleep_counter <- sleep_counter+1
 4   check if the primary replica is still alive
 5   if exception occurs,
 6     call handle_primary_Failure.
 7     call verify_primary
 8     register impl with the new primary
 9   if the primary is OK
10     if sleep_counter >=4
11       call verify_primary
12     sleep_counter <- 0
```

5.2.3 Handling Primary Failure

When the primary replica crashes or does not exist, a new primary will be elected from the backups. Backups will also update their reference to the new primary.

```
ALGORITHM handle_primary_failure()
 1   //Whenever exception occurs while invoking the primary replica,
 2   //this method should be called to handle all exceptions related
 3   //to primary replica provided that at least one registry exists.
 4
 5   //if there is no registry, this method can do nothing
 6   if no registry found yet
 7     return
 8
 9   //prepare a template. The template contains the group name
10   Prepare a template for lookup search for all replicas
11
12   //the lookup service may fail, too. Try every lookup registry
     //before giving up
13   for each entry in the vector of references to lookup registries
14     do a lookup for all replicas
15     if exception occurs
16   remove the registry from the vector of lookup references
17     exit the loop
18   if no replica is found
19   //Happens when no lookup exists, the whole system is down,
20   //stay alive for new lookup to appear.
21     return
22
23   //now found all replicas
```

```
24   miniID <-  registration ID of impl
25   miniRep<- impl
26   for each replica rep found
27     rep_id=rep's registration ID
28     if rep is primary //found an existing primary replica
29       set primary reference to rep
30       return
31     else if miniID > rep_id
32       miniID <- rep_id
33       miniRep<- rep
34
35   //Now no primary replica found
36   //And has found the replica with minimum registration ID,
37   //
38   If Rmin is not equal to the ID of impl
39     set the reference to primary as miniRep
40   If Rmin is equal to the ID of impl
41     register impl as primary with every registry
```

As we can see, the one with minimum registration ID will be elected as new primary. This eliminates the competition of backups to become a primary. Since registration IDs are of type long long (up to 128 bits) random strings, this helps to distribute primary task schedulers for different tasks into different hosts uniformly, which will balance the load on these hosts.

5.2.4 Verification

In a real network, there might be various issues to be considered, such as network latency and short, temporary loss of connectivity. Two replicas could join the group and announce as the primary replica at the same time. It could also happen that the primary expires, and backups fail to detect it, therefore no primary is registered. To avoid these problems, a periodic verification is conducted to make sure there is one and only primary replica, driven by the ping thread.

```
ALGORITHM verify_primary()
```

```
1   //check if the current primary replica is registered
2   //try every registry before giving up
3   found<-false
4   for each registry reg
5     do a lookup for primary_id
6     if exception occurs
7       remove reg
8       continue
9     if the replica is not found
10      continue //maybe has not registered with a new registry
11    if the replica is found
12      found <- true
13      exit the loop
14  //end of for loop
15
16  if(found == false)
17    primary_ref<-null
18    call handle_primary_failure to find a primary
19    return
20
21  //at this point, the current primary is registered
22  for each registry reg
23    do a lookup for all primary replicas
24    if exception occurs
25      remove reg
26      continue
27    if no primary found
28      continue //maybe this is a new registry
29  exit the loop //if everything is OK
30  //end of for loop
31
32  if only one primary replica is found
33    return
34  if no primary replica is found
35    primary_ref<-null
36    call handle_primary_failure
37    return
38
39  //at this point, more than one primary replica is found
40  //select the one with minimum ID as the primary
41
42  miniID <- primary_id
43  miniRep <- primary_ref
```

```
44   for each found primary replica rep
45     check if rep is still primary
46     rep_id <- the ID of rep
47     if rep is not primary
48       continue
49     if rep is primary
50       if miniID > rep_id
51         miniID <- rep_id
52         miniRep <- rep
53
54   if miniID is equal to primary_id
55     return
56   //at this point, miniID < primary_id
57   primary_ref <- miniRep
58   primary_id <- miniID
59
```

5.3 Replica Locator

This class is provided for convenience. Clients can use this class to locate a reference to a replica group. The reference is actually the reference to the primary replica of the group. The replica locator class has some similarities with the replica manager in that both need to first locate registries, then obtain reference to the primary replica. However, since the replica locator does not manage any replicas, its implementation is much simpler. Like the replica manager class, it also has the following important fields.

```
primary_ref : the reference to the primary replica
primary_id : the registration ID of the primary replica
```

Here is an example on how to use this class.

```
set criterion with group name
ref <- group reference
use ref to invoke methods
```

However, this is not the best way to use ReplicaLocator. Since the primary replica of the group may crashes during the invocation, the replica locator should be given

more chances to find a new valid group reference. The code below allows 3 failures while invoking a group.

```
 1   set criterion with group name
 2   count<-0
 3   if count >= 3
 4     exit loop
 5   get group reference
 6   use ref to invoke methods
 7   if exception occurs
 8     call handle_failure
 9     call verify
10     count <- count+1
11     go back to 3
```

5.3.1 GET REFERENCE TO A REPLICA GROUP

This method is very simple. It returns the value of the `primary_ref` attribute.

```
ALGORITHM get_reference
return primary_ref
```

5.3.2 SET CRITERION – WHICH GROUP TO LOCATE

This method allows clients to specify which group to locate. A group name is passed in as a parameter.

```
ALGORITHM set_criterion(group name)
 1   //initialization
 2   groupName <-  group name
 3   primary_ref <- null
 4   primary_id <- null
 5
 6   //try to find a reference to the primary
 7   call handle_failure to find a primary
 8   call verify to make sure this is the right primary
 9
10   //if no primary found, wait until it appear
11   loop for ever until primary_ref is not null
```

```
12     call handle_failure
13     call verify
14     if primary_ref is null
15     sleep for 3 seconds //there might be no such a group yet
16       //wait a while and check again
17     if primary_ref is not null //locate a primary
18       exit loop
19   //end of loop
20
```

5.3.3 HANDLING FAILURES

It is possible that the primary crashes during invocation. In this case, clients should call the `handle_failure` method of the replica locator and invoke the group again. The number of trials allowed can be specified by clients. After this number of trials, the group is unable to perform the requested service. This method is responsible to find a new primary replica.

```
ALGORITHM handle_failure
1
2   if no registry is found yet, loop for ever until found one
3   //at this point, there is at least one registry
4   for each registry reg
5     do a lookup for all primary replica
6     if exception occurs
7   remove the registry
8       continue
9     if no primary found
10  continue //try next replica
11    if a primary replica is found
12  primary_ref <- this replica
13  primary_id <- this replica's ID
14  //end of loop
15
```

5.3.4 PRIMARY VERIFICATION

This method is used to verify that this is the correct primary replica. Its algorithm is exactly the same as the `verify_primary` of the replica manager class.

5.3.5 PINGING THE PRIMARY REPLICA

Depending on the frequency of invocation, the replica locator may also start a ping thread to maintain an up-to-date reference to the primary, as the replica manager does. Again, it is very similar to the ping thread of replica manager, except that it does not register any replica with the primary.

5.4 Discussions

The main purpose of this model is to tolerate failures that might occur at any time. There are mainly three types of failures: network latency, expiration latency of registration and message losses. These issues may cause the system be unstable for a short period of time. The system of replicas may be in one of the following three types of states.

1. No primary replica exists.
2. Only one primary is registered.
3. More than one primary is registered.

In a steady stage, there is only one primary replica registered.

5.4.1 No primary replica exists or is registered

This might happen at the initialization of the system, or when the primary replica crashes. In the first case, a null pointer exception is thrown, and a remote exception is thrown in the second case. The `handle_primary_failure` method is then invoked

to find a new primary replica. And this system will soon (within two ping periods) enter a steady state with only one primary replica.

If there is no registry in the system, then no replica can become the primary. The ping threads will keep replicas alive, and once a registry enters the system, all replicas register themselves and then a primary replica is established.

5.4.2 Two or more registered primary replicas exist

Due to network latency, two or more replicas may find no primary and register themselves as primary at about the same time. The solution is to have a verification periodically, e.g. every 5 ping periods, which is done in the method `verify_primary`. If it turns out that only one primary is running, then the verification stops since it is what it should be. If there are more than one primary, then find the one with minimum registration ID and set reference to it as primary. In this way, the conflict is solved within 5 ping periods which is very short as compared to the running time of the system.

Chapter 6

Experiments

In this chapter, we will use our fault tolerant ORBWork (ORBWork$_{\mathrm{FT}}$) to show a series of examples. We will see their fault-tolerant deployment and how they tolerate breakups. Any components could crash at any time. Now, we will see, step by step, how the system responds to these fatal failures. Although servers such as the worklist servers and the ORBWork servers are all replicated actively, i.e., all replicas are equivalent, when clients contact these servers, the first one on the lookup list is the first one contacted. These first servers are denoted as *Selected*. An optimization is to contact these servers in a round-robin fashion, so each replica has a chance to be selected as the first, no matter where it is in the lookup list.

6.1 Setup of the Enactment System

As the first step, we started the servers on the hosts listed in Table 6.1. The starting order was not important. The system configuration is shown in table 6.1. Notice that every server was replicated. Most of them had two replicas, and the worklist server had three replicas. In general, the number of replicas for each server is arbitrary, but it should be at least one.

Host Name	Running Servers
jekyll	ORBWork Manager Worklist Server
greensboro	ORBWork Server Data Server
fargo	ORBWork Server Monitor Server (primary)
bainbridge	ORBWork Manager (primary) Worklist Server
cumming	Worklist Server (Selected) Data Server Monitor Server (backup)
chamblee	Registry Server
gemini	Registry Server

Table 6.1: The initial system configuration

6.2 SimpleFlow Workflow

We then installed the workflow system SimpleFlow and created several instances SimpleFlow0 and SimpleFlow1. On station Start, instance SimpleFlow0 was processed. The output of ORBWork servers indicated that the one on *greensboro* was selected as the first ORBWork server, which meant that the ORBWork manager always contacted this ORBWork server before it contacted the one on *fargo*. Therefore, all task schedulers were installed on *greensboro* before they were installed on *fargo*. Therefore, most primary task schedulers were residing on *greensboro*. Of course, as we mentioned earlier, the ORBWork manager could choose in a round-robin fashion an ORBWork server to install the first, most probably the primary task scheduler. From the output, we see that the following distribution of schedulers and task managers as shown in table 6.2

server	residing host of primary	residing host of backup
SimpleFlow.scd	fargo	greensboro
SimpleFlowManager	greensboro	fargo
Start.scd	greensboro	fargo
Process.scd	greensboro	fargo
Stop.scd	greensboro	fargo

Table 6.2: The distribution of task schedulers and managers

Host Name	Running Servers
jekyll	ORBWork Manager Worklist Server
greensboro	ORBWork Server Data Server
fargo	ORBWork Server Monitor Server (primary)
bainbridge	ORBWork Manager (primary) Worklist Server
cumming	Worklist Server (Selected) Data Server Monitor Server (backup)
chamblee	Registry Server

Table 6.3: The system configuration after a registry server crashes

6.3 Registry Server Failure

We killed the registry server on *gemini*, and then processed the instance SimpleFlow1 on station Start. There was no noticeable abnormal behavior at all. So the system tolerated registry crashes. The system configuration is shown in table 6.3

6.4 Monitor Server Failure

We killed the monitor server on *fargo* which was the primary one, and processed SimpleFlow0 all the way through. Again, there was no noticeable abnormal behavior.

Host Name	Running Servers
jekyll	ORBWork Manager Worklist Server
greensboro	ORBWork Server (Selected) Data Server
fargo	ORBWork Server
bainbridge	ORBWork Manager (primary) Worklist Server
cumming	Worklist Server Data Server Monitor Server (primary)
chamblee	Registry Server

Table 6.4: The system configuration after a monitor crashes

So the system tolerated monitor failures. The system configuration is as in table 6.4. Notice that the monitor on *cumming* was now the primary monitor server.

6.5 Data Server Failure

We killed the data server on *cumming*, and created a new instance SimpleFlow2 and processed it on station Start. No abnormal behavior was found. The system tolerated the crashes of data servers. The system configuration is shown in table 6.5.

6.6 ORBWork Manager Failure

We killed the ORBWork Manager (the primary one) on *bainbridge*. The web browser was then notified of the failure of the ORBWork Manager server. We connected to http://jekyll:9002. No changes was found. Two more instances SimpleFlow3 and SimpleFlow4 were created. On station Start, instance SimpleFlow3 was processed. Everything went all right. So the system tolerated the crashes of ORBWork Managers. The system configuration is shown in table 6.6.

Host Name	Running Servers
jekyll	ORBWork Manager Worklist Server
greensboro	ORBWork Server (Selected) Data Server
fargo	ORBWork Server
bainbridge	ORBWork Manager (primary) Worklist Server
cumming	Worklist Server (Selected) Monitor Server (primary)
chamblee	Registry Server

Table 6.5: The system configuration after a data server crashes

Host Name	Running Servers
jekyll	ORBWork Manager (primary) Worklist Server
greensboro	ORBWork Server (Selected) Data Server
fargo	ORBWork Server
bainbridge	Worklist Server
cumming	Worklist Server (Selected) Monitor Server (primary)
chamblee	Registry Server

Table 6.6: The system configuration after an ORBWork manager crashes

server	residing host of primary
SimpleFlow.scd	fargo
SimpleFlowManager	fargo
Start.scd	fargo
Process.scd	fargo
Stop.scd	fargo

Table 6.7: The distribution of task schedulers after an ORBWork server crashes

6.7 ORBWork Server and task scheduler failure

On station Process, instance SimpleFlow1 was processed. During the processing, the ORBWork server on *greensboro* was killed. Since task schedulers reside on the ORBWork server, all task schedulers on *greensboro* crashed. The distribution of task schedulers are shown in table 6.7.

SimpleFlow1 was removed from station Process and appeared on station Stop, despite the fact that the primary task scheduler Process crashed before SimpleFlow1 was processed successfully. This was exactly what we expected. We created two new instances SimpleFlow5 and SimpleFlow6, and then processed SimpleFlow5 all the way through. This indicated that the system tolerated completely crashes of ORBWork servers since task schedulers on *fargo* continued to function. In addition, it tolerated completely crashes of task schedulers. The system configuration is now as Table 6.8.

6.8 Worklist Server Failure

We killed the worklist server on *cumming* which was the first worklist server. In the web browser, we refreshed the page http://jekyll:9002, then the link was redirected to the worklist server http://jekyll:9009. The system configuration is as table 6.9.

Host Name	Running Servers
jekyll	ORBWork Manager (primary) Worklist Server
greensboro	Data Server
fargo	ORBWork Server
bainbridge	Worklist Server
cumming	Worklist Server (Selected) Monitor Server (primary)
chamblee	Registry Server

Table 6.8: The system configuration after an ORBWork server crashes

Host Name	Running Servers
jekyll	ORBWork Manager (primary) Worklist Server
greensboro	Data Server
fargo	ORBWork Server
bainbridge	Worklist Server
cumming	Worklist Server (Selected) Monitor Server (primary)
chamblee	Registry Server

Table 6.9: The system configuration after a worklist server crashes

On station Start, we successfuly processed SimpleFlow6. We created a new instance SimpleFlow7, and found that SimpleFlow7 were also processed all the way through. This indicated that the system could tolerate completely the crashes of Worklist servers.

6.9 SUMMARY

With the presence of registry servers, the system configuration becomes very flexible. Only the port numbers, the monitor log directory and the path to the shell command *sed* need to be specified. The servers can be started on arbitrary hosts in the arbitrary order, whereas the original implementation required strict starting order, and that all servers be installed on a single host that is specified in the property file.

Our system completely tolerates crashes of the registry servers, the monitor servers, the data servers, the ORBWork$_{\mathrm{FT}}$ managers, the ORBWork$_{\mathrm{FT}}$ servers, worklist servers and the task schedulers.

Chapter 7

Conclusions and future work

Our experiment results show that replication can dramatically improve the fault tolerance of workflow systems. The system configuration becomes highly flexible, and it tolerates sequencing fault at system startup. Our system completely tolerates crashes of the registry servers, the monitor servers, the data servers, the $\text{ORBWork}_{\text{FT}}$ managers, the $\text{ORBWork}_{\text{FT}}$ servers, the task schedulers, and the worklist servers.

The system also allows increase of the number of the monitors, the ORBWork managers and the registry servers, dynamically at runtime. It is expected that other servers will be allowed to join the system during runtime if new replicas are updated with full logs upon becoming a member, which is not fully implemented yet.

Optimization is also needed to enhance the performance while retaining fault tolerance, including applying asynchrony and using reliable multicast in replica group coordination. With multicast, task schedulers will be distributed evenly among ORBWork servers since commands from ORBWork managers arrive at ORBWork servers approximately at the same time.

The remote event mechanism supplied by Jini could also be exploited to reduce the network overheads in the current implementation. With remote event notification, backup replicas will be notified when the primary fails, which eliminates the need for backups to inquire frequently the status of the primary and, therefore, reduces the network overheads dramatically.

The replica manager and locator classes may be further optimized by providing static methods and fields that are shared among all instances in a JVM. In the current

implementation, each replica manager/locator have its own multicast message and listener, which is easier in implementation and wasteful in resources. By this sharing, only one multicast message and one listener are needed in a single JVM, which again cuts the overhead dramatically. With these optimizations, we expect that the response time of the system will decrease significantly.

Besides the optimizations above, the future work also include providing graphic tools for configuring the system, such as browsing replicas' distribution, specifying the number of replicas for individual components and their residing hosts.

Bibliography

[ABH00] Aalst, W., Barros, A., Hofstede, A. and Kiepuszewski, B. (2000) *Advanced Workflow Patterns* Conference on Cooperative Information System, 2000

[AH00] Alonso, G.; Hagen, C.; et al (2000) *Enhancing the Fault Tolerance of Workflow Management Systems.* IEEE Concurrency, 2000.

[Alo00] Alonso, G.; et al (2000) *Distributed Processing over Stand-Alone Systems and Applications.* 23rd Int'l Conf. Very Large Databases, Morgan Kaufmann, San Francisco, 1997, pp.575-579.

[Bac99] Bacon, J. (1999) *Special Issue on Workflow Management Systems.* IEEE Concurrency, July-Sep. 1999

[CDK01] Coulouris, G., Dollimore, J., Kindberg, T. (2001)*Distributed Systems.* 3rd ed., published by Addison-Wesley and Pearson Education

[DKM96] Das, S., Kochut, K., Miller, J., Sheth, A., Worah D. (1996) URL: http://citeseer.nj.nec.com/das96orbwork.html

[GHS95] Georgakopoulos, D., Hornick, M., and Sheth, A. (1995)*An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure.* Distributed and Parallel Databases, 3, 119-153(1995)

[HA98] Hagen, C.; Alonso, G. (1998) *"Flexible Exception Handling in the OPERA Process Support System" 18th Int'l Conf. Distributed Computing Systems.* IEEE Computer Soc. Press, Los Alamitos, Calif., 1998, pp. 526-533

[Hol94] Hollinsworth, D. (1994) *The Workflow Reference Model.* Workflow Management Coalition, Tech. Report TC00-1003, Dec. 1994; www.wfmc.org.

[Jini] Sun's online support for Jini, URL:

http://www.sun.com/software/jini/overview/index.html

[KAG95] Kamath, M., Alonso, G., Gunthor, R. and Mohan, C. (1995) *Providing High Availability in Very Large Workflow Management Systems*, Research Report RJ9967, IBM Almaden Research Center, July 1995

[KFS99] Kang, M., Froscher, J., Sheth, A., Kochut, K., Miller, J. (1999) *A Multilevel Secure Workflow Management System* http://chacs.nrl.navy.mil/publications/CHACS/1999/1999kang-CAISE99.pdf

[Koc98] Kochut, K. (1998) *METEOR Model Version 3.* Technical Report, University of Georgia

[KS95] Krishnakumar, N. and Sheth, A. (1995) *Managing Heterogeneous Multisystem Tasks to Support Enterprise-wide Operations.* Distributed and Parallel Databases, 3(2):155-186, April 1995

[KSM99] Kochut, K., Sheth, A. and Miller, J. (1999) *Optimizing Workflow*, Component Strategies, Vol. 1, No. 9, 1999, pp. 45-57

[Ley01] Leymann, F. (2001) *Web Services Flow Language (WSFL 1.0)* IBM Software Group

[LSK00] Luo, Z., Sheth, A., Kochut, K. and Miller, J. (2000) *Exception Handling in Workflow Systems.* Applied Intelligence, Volume 13, Number 2, September/October, 2000, pp.125-147

[LSP82] Lamport, L., Shostak,R. and Pease M. (1982) *Byzantine Generals Problem* ACM Transactions Programming Languages and Systems, Vol. 4, No. 3, pp. 382-401

[MAG95] Mohan, C., Alonso, G., Gunthor, R., Kamath, M. and Reinwald, B. (1995) *An Overview of the Exotica Research Project on Workflow Management Systems*, Proc. 6th Int'l Workshop on High Performance Transaction Systems, Asilomar, 9/95

[MOK99] Muhlberger, R., Orlowska, M. and Kiepuszewski, B. (1999) *Backward Step: the Right Direction for Production Workflow Systems*, Proceedings of the Tenth Australasian Database Conference, Auckland, New Zealand, January 18-21 1999. Springer-Verlag, Singapore.

[New00] Newmarch, J. (2000), URL:

http://pandonia.canberra.edu.au/java/jini/tutorial/Overview.xml

[RMI] Sun's online tutorial for RMI, URL:

http://java.sun.com/products/jdk/rmi/

[SK98] Sheth, A. and Kochut, K. (1998) *Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems.* Workflow Management Systems and Interoperability, A. Dogac et al[Eds], Springer Verlag, 1998, pp.35-60.

[SWK97] Sheth, A., Worah, D., Kochut, K., Miller, J., Zheng, K., Palaniswami, D., Das, S. (1997) *The METEOR workflow management system and its use in prototyping significant healthcare applications.* Proceedings 1997 Toward and Electronic Patient Record Conference (TEPR '97). Vol. 2, Nashville, TN pp. 267-278

[Tri00] Tripathy, S. (2000) *WFTP: Design and a RMI Implementation of a Workflow Transport Protocol for ORBWork* Master thesis, 2000, University of Georgia.

[WMC95] Workflow Management Coalition's reference model, URL:

http://www.wfmc.org/standards/docs/tc003v11.pdf

[Wor97] Worah, D. (1997) *Error Handling and Recovery for the ORBWork Workflow Enactment Service in METEOR* Master thesis, University of Georgia

[WPS00] Wiesmann, M., Pedone, F., Schiper A., et al, (2000) *Understanding replication in databases and distributed systems* Proceedings 20th International Conference on Distributed Computing Systems(ICDCS'2000), Taipei, China, IEEE

[WSR98] Wheater, S., Shrivastava, S. and Ranno F. (1998) *A CORBA Compliant Transactional Workflow System for Internet Applications* Preceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Proceeding (MIDDLEWARE'98), The Lake District, England, September 15-18, 1998