Decentralized Data Dependency Analysis for Concurrent Process Execution

by

Ziao Liu, B.E.

A Thesis

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Approved

December, 2009

Acknowledgments

I am heartily thankful to my academic advisor, Dr. Susan Urban, whose encouragement, guidance and support from the very beginning of my graduate study to the end of my master thesis has made all this happen.

I greatly appreciate Le Gao for his opinions and help on the research and this thesis.

I would like to thank all my friends for being so supportive.

This work is dedicated to my parents.

TABLE OF CONTENTS

ABSTRACT

In processes composed of Web Services, interleaved access to data between service executions of concurrent processes can potentially cause data inconsistency problems. If a process fails, data items modified by the recovery of a failed process may affect other processes that are concurrently executing and have accessed the same data items. The results of this research present a decentralized approach to analyzing data dependencies among concurrently executing processes in a service-oriented environment. The decentralized approach is an extension of past research with the DeltaGrid project that analyzes data changes captured from service executions to identify processes that are dependent on a failed process based on data access patterns. The results of this research have defined algorithms that allow multiple process execution engines to share information about data dependencies. Process Execution Agents (PEXAs) have been defined that control the execution of processes and build local delta object schedules. Process execution histories are then enhanced with control information that allows the construction of data dependency graphs to be distributed among multiple PEXAs by sharing data dependency information, This research has explored a lazy algorithm that constructs distributed process dependency graphs upon the failure of a process. The research has also explored an eager algorithm that dynamically constructs process dependency graphs for all executing process so that dependency graphs are available as soon as a failure occurs. The work includes an analysis of performance characteristics of the algorithms. The results of this research represent an initial step towards the development of distributed, process-aware execution environments that can support more intelligent ways of monitoring failures, detecting dependencies, and responding to failures and exceptional conditions in an environment that cannot conform to traditional data locking protocols.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Service-Oriented Computing (SOC) is a new computing paradigm that utilizes web services as its basic components, supporting the demand for higher interoperability, scalability, and flexibility in software development practice (Singh and Huhns, 2005). Web service components are loosely-coupled, platform independent and distributed in an SOC environment to work collaboratively. Hence, distributed software is developed using service composition, sometimes creating long-running computational elements.

Since each service in a process is autonomous and platform-independent, the commit of a service execution is controlled by the residing service instead of the global process. As a result, distributed processes composed of services do not execute as traditional transactions as in centralized and distributed database applications. The concept of serializability is too strong for concurrently executing services to conform to global transaction semantics as one process. As a result, ACID (atomicity, consistency, isolation and durability) properties and traditional concurrency control mechanisms are not generally suitable for this environment, since a process cannot afford to block individual services to ensure a commit of the global process (Mikalsen et al., 2002). Hence, dirty writes and dirty reads are inevitable since a service can commit before a process completes. This interleaved access to data between service executions of concurrent processes can potentially cause data inconsistency problems. If a process fails, data items modified by this failed process may affect other processes which are concurrently executing and have accessed the same data items. As a result, there exists data dependencies between concurrently executing processes. Information about data dependencies can potentially be used to enhance recovery procedures and to provide more intelligent ways to address data consistency issues.

Several recent research endeavors have focused on the recovery of processes in a service-oriented environment. One such project that has addressed the recovery of dependent processes is the DeltaGrid Project (Xiao, 2006; Xiao and Urban, 2008a; Xiao and Urban, 2008b). In the DeltaGrid system, services are referred to as Delta-Enabled Grid Service (DEGS) (Urban et al., 2009a), and are extended to capture incremental data changes, known as *deltas*. A subsystem known as the Process History Capture System (PHCS) is created to receive, store, and analyze deltas. The DeltaGrid system is capable of providing a limited form of rollback (known as Delta-Enabled Rollback) as well as compensation and contingency procedures. The merged deltas are also used to analyze dependencies when a process fails and to invoke a recovery process that uses user-defined rules to determine forward or backward recovery actions for dependent processes (Xiao, 2006; Xiao and Urban, 2008a; Xiao and Urban, 2008b).

In the DeltaGrid project, a centralized PHCS merges multiple streams of deltas to create a time-sequenced global schedule of data changes. Distributed deltas in the execution environment are transmitted, stored at the central delta repository, and used to create execution context and a global schedule. The work in (Urban et al., 2009a; Xiao and Urban, 2008a) demonstrated the feasibility of collecting data changes for distributed service execution and analyzing data dependencies to identify how one process can potentially affect other processes, especially during failure and recovery activities. The research also demonstrated the overhead associated with the centralized approach to the analysis of data dependency.

The purpose of this research has been to investigate a decentralized approach to data dependency analysis in a service-oriented environment. In particular, this research has investigated the concept of Process Execution Agents (PEXAs) and the manner in which multiple PEXAs communicate to discover data dependencies that can be used to support recovery activities. PEXAs are responsible for controlling the execution of processes that are composed of web services. PEXAs are associated with specific distributed sites and are also responsible for capturing and exchanging information with other PEXAs about the data changes that occur at those sites in the context of service executions.

This research defines the functionality of PEXAs and also describes the data structures and communication mechanisms that are used to achieve a decentralized approach to the analysis of data dependencies and the construction of distributed process dependency graphs. Two different decentralized algorithms for data dependency analysis have been developed. One approach, known as the *lazy algorithm*, defers the analysis of data dependencies until the failure of a process. When a process fails, PEXAs communicate to construct a distributed process dependency graph that is used to control recovery activities. The other approach, known as the *eager algorithm*, constructs distributed process dependency graphs during process execution so that dependency information is already known at the time of process failure. Performance results of the lazy approach indicate an inverse relationship between local graph construction time and the percentage of externally executed operations, with graph construction time also affected by the percentage of dependencies. Higher failure rates also correlate with increased graph construction time. Performance results from the eager approach shows that adding nodes to the graphs does not consume a large amount of time with an increase in the level of the process execution.

The decentralized algorithms presented in this thesis eliminate the bottleneck reported in (Urban et al., 2009a; Xiao and Urban, 2009; Xiao, 2006) of forwarding all data changes to a central point for analysis. More importantly, the distributed delta object schedule and decentralized data dependency analysis algorithms represent a new way of integrating existing transaction processing theories with execution platforms that can be used to address data consistency issues for concurrent process execution in service-oriented environments, providing more dynamic and intelligent ways of monitoring failures, detecting dependencies, and responding to failures and exceptional conditions.

In the remainder of this thesis, Chapter 2 presents related work. Chapter 3 provides an overview of the DeltaGrid project on which this research is based. Chapter 4 then outlines the functionality of PEXAs with an illustration of the challenges associated with decentralized data dependency analysis. In Chapter 5, the lazy and eager algorithms are presented for decentralized data dependency analysis and propagation of recovery. Chapter 6 analyzes performance characteristics of the algorithms. The thesis concludes in Chapter 7 with a summary and discussion of future research directions.

CHAPTER II

RELATED WORK

This chapter discusses related work. Section 2.1 summarizes work with advanced transaction models and service composition. Section 2.2 outlines past work with transactional workflows. Section 2.3 presents issues related to the transactional aspects of service composition.

2.1 Advanced Transaction Models

Advanced Transactional Models (ATMs) (Worah and Sheth, 1997) relax the traditional ACID properties and use of the two-phase commit protocol to provide functionalities such as compensation for backward recovery and contingency for forward recovery.

In the work of (Garcia-Molina and Salem, 1987), a mechanism is proposed to structure a long running process as a saga consisting of many ordered smaller tasks. Each of these smaller tasks conforms to ACID properties, so the tasks from different processes are allowed to execute as interleaved operations. Furthermore, there could be nested Sagas which means that each task can have a sequence of smaller tasks that adhere to ACID properties.

Traditional ACID properties require all or nothing execution. Hence, when a transaction aborts after a failure, everything has to be rolled back. For each task in a saga, there should be a compensator (Garcia-Molina and Salem, 1987) to reverse the original task. The compensator is an execution that logically removes the results of the failed task. Therefore, to abort a saga, the system aborts the current active task and executes the compensators for each task in reverse order to remove any actions performed by an each task of the saga.

The flexible transaction model (Ansari et al., 1992) consists of three components: a set of sub-transactions, a set of intra-transaction execution dependencies and a set of acceptable states. A flexible transaction has a set of sub-transactions, each of which is a logical step to do some operations. Since a sub-transaction can be either compensable or noncompensable, different constraints are applied. A compensable sub-transaction can commit locally after finishing its operations, and its results are visible to other transactions or sub-transaction. If something goes wrong, the sub-transaction can be compensated. However, the noncompensable sub-transactions must wait for the commit by the global transaction, so its results can be made visible to others.

The set of intra-transaction execution dependencies defines the execution semantics. A sub-transaction has an execution dependency as a condition to start, to resume or to terminate a sub-transaction. Execution dependencies could be based on the execution state, on the output of other transactions, or on time, so that operations can be grouped to form one or more sub-transactions.

The set of acceptable states defines the conditions for the success of a flexible transaction, with a combination of the execution states of sub-transactions. Each sub-transaction has an acceptable state. If the acceptable state is reached, the sub-transaction can terminate successfully without additional sub-transactions. Therefore, during the execution of a flexible transaction, all sub-transactions are scheduled according to their execution dependencies. Then the scheduler determines if an acceptable state is reached whenever the execution state of a sub-transaction is changed. If an acceptable state is reached, the sub-transaction is ready to commit, and the acceptable state is used to decide whether to commit, abort or compensate this sub-transaction. Otherwise, the sub-transaction has to be aborted or compensated according to the execution state.

The multilevel transaction model (Weikum, 1991) allows more concurrency in the execution of independent transactions. A transaction is decomposed into a nested set of sub-transactions at different levels and the individual sub-transactions can unconditionally commit before the whole multilevel transaction commits. An operation at some level usually creates its sub-transactions in the next level as child sub-transactions and the parent transaction waits until the child sub-transactions commit. A sequence of abstractions is defined for multilevel transaction to access the database. At the lowest level, sub-transactions are read or write operations over sets of pages. At the next higher level, the abstraction of tuples are accessed using SQL statements. At an even higher level, some interfaces specifically defined can be seen (Weikum, 1991).

These techniques relax the ACID properties and introduce the concept of compensation for the decomposed sub-transactions. They also define rule-based conditions for execution. However, they do not support isolation of data and do not address recovery for dependent transactions in a loosely-coupled environment.

2.2 Transactional Workflows

The term Transactional Workflows is introduced to clearly recognize the relevance of transactions to workflows. Transaction workflows involve the coordinated execution of multiple related tasks that support access to heterogeneous, autonomous, and distributed data through the use of selected transactional properties for individual tasks or entire workflows (Worah and Sheth, 1997). The ConTract Model is one of many models addressing transactional workflows (Wachter and Reuter, 1992).

The ConTract Model supports the correct execution of non-atomic, long-lived applications with application-dependent consistency constraints. The model provides a mechanism for grouping transactions into a multi-transaction activity. A ConTract consists of a set of predefined actions (steps) and an explicitly specified execution plan (script). The ConTract Model provides compensation for backward recovery, and user-defined consistency through the specification of pre-conditions or post-conditions for steps. After the execution of each step, the ConTract Model will release locks and if failure occurs, the ConTract Model will semantically undo the effect of completed steps. The pre-/post-condition guarantees the user-defined way of specifying correctness criteria. However, conditions cannot be checked during the execution, and the execution of its compensation is not flexible enough to support it in different contexts.

The work in (Eder and Liebhart, 1995) introduced the workflow activity model (WAMO). WAMO enables the workflow designer to model complex business processes in a simple and reliable way. A complex business process or workflow is decomposed into smaller work units, known as activities, which consist of pre-existing tasks and automatic exception handling for reliable flow control. In this model, a workflow consists of one or more activities, forms and agents. An activity indicates any abstract description of work units in the business process. A form is a data repository or container to store relevant data. An agent is a processing entity to perform some execution of activities. An activity may consist of multiple other activities as its steps. Furthermore, activities may be reused by other activities. Hence, new workflows are allowed to be composed of predefined activities. Five simple but powerful control structures are provided to flexibly compose a workflow: Sequence, Ranked Choice, Free Choice, Parallel and Nesting. After each execution of an activity, the execution state will also be used to control activities of a workflow.

In the Correct and Reliable Execution of Workflows (CREW) project (Kamath and Ramamritham, 1998), the correctness requirements and other constraints are specified for the workflow executions based on the previous work on transactional workflows. A workflow executes on multiple steps, where a step is triggered by the completion of one or more previous steps, or the occurrence of specific events. The rules, events or conditions predefined will be used to dynamically generate the rule sets to manage the execution of workflows. A mechanism is proposed for the handling of failures to eliminate unnecessary compensations and re-execution of steps. Depending on whether the previous execution of steps is acceptable, complete compensation and re-execution, or partial compensation and incremental re-execution is used to undo the effects. Therefore, CREW makes the execution of workflows more dynamic by the use of dynamic rule set. The handling of failures and exceptions can be better managed during execution.

The METEOR model (Wachter and Reuter, 1992) integrates many approaches from the transactional workflows models. A METEOR model includes four components, processing entities and their interfaces, tasks, task managers and workflow schedulers. A processing entity is responsible for the completion of a task, which could be performed by a user or application system. A task is a basic execution agent that performs some operations. The task manager takes control of each task and its interactions with the environment. The workflow scheduler is responsible for coordinating the execution of tasks within a workflow. Errors are defined and captured by this model.

The transactional workflows models have been used to define application specific and user-defined correctness, reliability and functional requirements within workflow executions. These models have improved the robustness of distributed transaction executions. Transactional workflows provide solutions to recover a failed transaction, but the recovery of one failed transaction could affect other concurrently executing transactions because the relaxation of atomicity and isolation is not considered. Furthermore, the execution dependencies and rule sets are static, which means that they are predefined for errors and foreseen exceptions.

2.3 Transactional Aspects of Service Composition

Web services are becoming more popular in the business-to-business computing environments. A Web service is defined in (Booth et al., 2005) as “a software system designed to support interoperable machine-to-machine interaction over a network”. A Web service uses the Web Service Definition Language (WSDL) to define its interface. The Simple Object Access Protocol (SOAP) messages in an XML format are exchanged, which is the basic interaction between services. Other web service standards that provide protocols and frameworks for web services are

WS-coordination (Cabrera et al., 2002a), WS-atomic transaction (Cabrera et al., 2002b) and WS-Business activity (Newcomer et al., 2006).

The WS-coordination specification (Cabrera et al., 2002a) defines an extensible framework for coordinating distributed computational units using a coordinator and a set of coordination protocols. Therefore, distributed applications can reach a consistent state by the coordination protocols. The coordination protocols are suitable for a wide variety of application activities, such as simple short-lived operations and complex long-lived business activities. This model defines a framework for a coordination service which consists of three component services: 1) an activation service with an operation that enables an application to create a coordination context; 2) a registration service with an operation that enables an application to register for coordination protocols; 3) and a coordination type-specific set of coordination protocols. The WS-coordination specification also provides for extensibility and flexibility as follows: the publication of new coordination protocols, the selection of a protocol from a coordination type, and the definition of extension elements that can be added to protocols and message flows.

The WS-transaction specification (Cabrera et al., 2002b) defines three specific agreement coordination protocols for the atomic transaction coordination type: completion, volatile two-phase commit (Volatile 2PC), and durable two-phase commit (Durable 2PC). The completion protocol initiates commitment processing. Based on each protocol's registered participants, the coordinator begins with Volatile 2PC, then proceeds through Durable 2PC. The final result is signaled to the initiator. The 2PC protocol coordinates registered participants to reach a commit or abort decision, and ensures that all participants are informed of the final result. Participants managing volatile resources such as memory should register for Volatile 2PC, and participants managing durable resources such as a database should register for Durable 2PC. Atomic transactions conform to an all-or-nothing property.

The WS-Business activity specification (Newcomer et al., 2006) defines the business activity coordination type which is used to coordinate with the extensible coordination framework described in the WS-Coordination specification above. Hence, business process and workflow systems are enabled to wrap their proprietary mechanisms and interoperate across trust boundaries and different vendor implementations.

Web Service Business Process Execution Language (Jordan et al., 2007), or WS-BPEL, is a standard to enable the interoperability and the integration between business processes by specifying interactions with web services. This standard defines all the elements to build heterogeneous and distributed applications. To define a business process, basic components need to be included, such as partner links, properties, correlation, activities, scopes, and handlers. A partner represents both a consumer of a service provided by the business process and a provider of a service to the business process. The definition of properties creates a unique name for a WS-BPEL process definition and associates it with an XML Schema type. Correlation is used to provide additional application-level mechanisms to match messages and conversations with the business process instances for which they are intended. Basic and structured activities are also modeled. Basic activities include receive (doing a blocking wait for a matching message to arrive), reply (sending a message in reply to a formerly received message), invoke (invoking a one-way or request-response operation), assign (updating the values of variables or partner links with new data), and validate (validating XML data stored in variables). Structured activities are used for control flow. The scope concept indicates a set of activities which could be basic or structured operations. There are also three kinds of handlers: event handlers for message events or time events, fault handlers for exceptional situations, and compensation handlers for undoing the effects of completed activities.

According to the WS-BPEL standard (Jordan et al., 2007), business processes are composed of web services. A business process cannot always conform to traditional concurrency control mechanisms and ACID properties since it is not reasonable to block individual services to wait for a commit of a global process. Isolation has to be relaxed since each locally residing service will decide the commit of an operation, and as a result, data dependencies are generated between global processes. So the recovery of a failed process might affect other concurrently executing processes. Therefore, the most critical issue in the transactional aspects of service composition is to guarantee correctness in this concurrent environment for failure recovery.

The techniques mentioned above do not deal with the data dependencies that exist between concurrent process executions. If a process fails, it is likely to make one process consistent by the recovery. But other business processes may not be consistent since there is no mechanism to detect and identify data dependencies between concurrent processes. Recent work such as the promises model in (Greenfield et al., 2007) and the reservation-based technique in (Zhao et al., 2005) propose new mechanisms to address the use of shard data in concurrent process execution environments. The promises model provides agreements between clients and the resources and defines conditions for the control of operations. The reservation protocol uses a reservation-based mechanism to coordinate transactions without using locking for data access.

The technique presented in this research dynamically analyzes write dependencies and potential read dependencies among concurrently executing processes by capturing data changes from distributed service executions and providing an intelligent, decentralized approach to discovering dependencies that can be used to enhance recovery techniques such as those described above.

CHAPTER III

OVERVIEW OF THE DELTA GRID PROJECT

The research described in this thesis builds on past work with the DeltaGrid project (Xiao, 2006; Xiao and Urban, 2008a; Xiao and Urban, 2008b; Xiao and Urban, 2009) and Delta-Enabled Grid Services (DEGS) (Urban et al., 2009). This chapter provides an overview of foundational work with the DeltaGrid project. Section 3.1 discusses DEGS and how they are used to generate information about data changes. Section 3.2 then describes process history capture system. Section 3.3 summarizes how deltas are used in the recovery process.

3.1 Delta Enabled Grid Service (DEGS)

A DEGS is a Grid Service that has been enhanced with an interface that provides access to the incremental data changes, known as *deltas*, that are associated with service execution in the context of globally executing processes. A DEGS uses an Open Grid Services Architecture Data Access and Integration (OGSA-DAI) Grid Data Service for database interaction and the Globus Toolkit to provide containers for the host of grid services. The OGSA-DAI (Foster et al., 2004) implements Java-based grid services which then can access and integrate data resources such as DB2, Oracle, SQL Server, and other major commercial database systems.

The database accessed by a DEGS captures deltas using capabilities provided by most commercial database systems. In (Urban et al., 2009a), triggers and Oracle Streams (Tumma, 2004) are used as a way to capture data changes(Oracle, 2005)(Oracle, 2005). Oracle Streams is a feature that monitors database redo logs for changes and publishes these changes to a queue to be used for data sharing.

Deltas captured over the source database are stored in a local delta repository. Deltas are then generated as a stream of XML data from the delta repository to the Process History Capture System (PHCS) of the DeltaGrid execution environment. The object delta structure that formats XML deltas is shown in Figure 1.

A DeltaObject contains information about the data items that have been changed by a service: className, oId and degsId. className indicates the name of the class/relation that is modified. oId is the unique identifier of the object/tuple instance and degsId is the identifier of the service that is modifying the data. A DeltaObject can have multiple Property objects. Each Property includes a propertyName as the name of the changed property, and one or more PropertyValue objects indicating the history of data values of an attribute. PropertyValue has a one-to-one mapping to a DataChange object, which has a processId, operationId and timestamp to indicate a specific process and its operation at a certain time when a PropertyValue object is created.

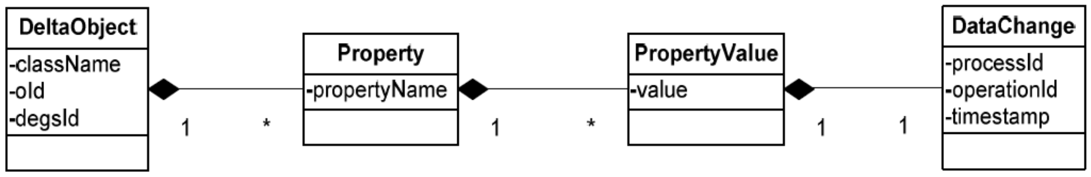


Figure 1: Object Delta Structure (Urban et al., 2009a)

As a grid service makes changes to data items in a database, the changes are packaged in an XML format according to the structure in Figure 1 and forwarded to the process history capture system.

3.2 Process History Capture System (PHCS)

The PHCS parses, analyzes, organizes, and records deltas for execution recovery. The PHCS is comprised of three layers: a delta storage layer that stores the deltas, a data access layer that provides interfaces for reading from and writing to the delta repository, and a service layer that receives and parses deltas sent through XML files (Xiao, 2006).

When receiving captured deltas, a complete execution history for distributed, concurrent processes is formed. The execution history includes deltas from distributed DEGSs and the process runtime context generated by the process execution engine.

Deltas are dynamically merged using timestamps as they arrive in the PHCS to create a time-ordered log of delta objects from distributed DEGS, which is called the *global delta object schedule*.

The global delta object schedule shows how concurrent processes interleave access to shared data items. The schedule supports recovery activities like the backward recovery of a completed service and also provides the basis for discovering data dependencies among processes.

An indexing structure is provided for the global delta object schedule using runtime information and deltas. This conceptual view of the global schedule obtains all the active processes and their operations from the process runtime information repository. The global schedule also contains a time-ordered list of node structures indicating the data that has been modified. A time-sequence index is established to retrieve a node by specifying the processes and operations. The node and time-sequence index is a one-to-one mapping, retrieving the delta repository for the delta objects that are stored into the global schedule as a key-value pair. When deltas arrive as XML files, they are extracted using the object delta structure and added into the global schedule organized through the indexing structure. Figure 2 shows the conceptual view of the global schedule as defined in (Xiao, 2006).

Data dependencies are used to identify concurrently executing processes that may be affected by the failure and recovery of a process that is accessing shared data. Using the global schedule, processes that are write dependent on a failed process can be detected. Write dependency exists between two processes if one process modifies the data objects that have been written by another process that has not yet committed (Xiao, 2006). Since the delta object schedule only contains information about modified data, potential read dependencies can only be derived from the process and operation context, and not from the delta repository. Potential read dependencies are handled in a conservative way by detecting services that have executed at the same site during the same time period. If a process fails, the data items modified by the failed process may affect other concurrently executing processes, thus creating data consistency issues in the execution environment. Therefore, write and read dependencies can be identified based on the definitions and then used in the recovery process.

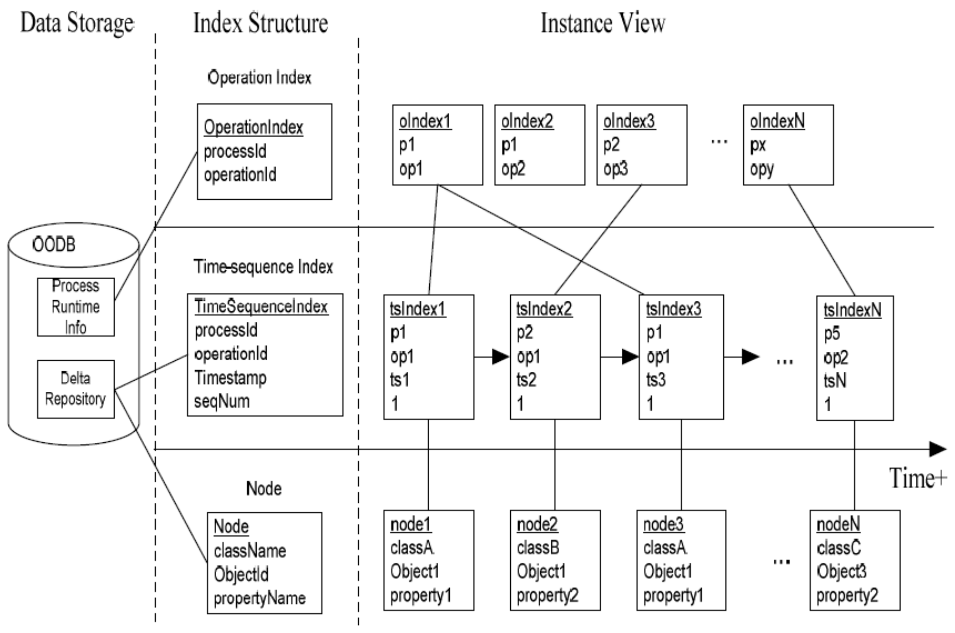


Figure 2: Conceptual View of the Global Schedule (Xiao, 2006)

3.3 Using Deltas in the Recovery Process

Besides DEGS and PHCS, the DeltaGrid project introduced the use of process interference rules (PIRs) (Xiao, 2006). Process Interference Rules are active rules that query the data changes from a failed process and its potential read and write dependent processes, using application semantics to determine if the recovery of a failed process has an effect on active processes that are read and/or write dependent on the failed process (Xiao and Urban, 2009). Process interference rules retrieve the delta values of the global execution history, using user-defined semantics to determine how to deal with an affected process.

Figure 3 shows the rule structure of a process interference rule. There are four elements for a process interference rule: event, define, condition and action. When a process fails, all processes that are read and/or write dependent on the failed process are identified. Each dependent process generates a failureRecoveryEvent that triggers a PIR. The define element is used to specify the data items that are monitored by the rules. The condition element is used to test an application-specific condition that determines if recovery of a dependent process is required. The action element contains a list of recovery commands. If the condition is satisfied, the commands specified in the action element will be executed.

create rule	ruleName
event	failureRecoveryEvent
define	[viewName as <OQL expression>]
condition	[when condition]
action	recovery commands

Figure 3: Process Interference Rule Structure (Xiao and Urban, 2008a)

In summary, DEGSs that are executing concurrently send deltas to the centralized PHCS from multiple sites. Deltas are constructed into the global schedule through the indexing structure and stored in the delta repository together with runtime information. If a process fails, the processes that are write dependent or potentially read dependent on the failed process will be detected using the procedures mentioned earlier to get the dependency information. At this time, if there is a PIR for a process, the process is suspended and the PIR is checked for user-defined rules that determine whether to recover the affected process.

The DeltaGrid project demonstrated the feasibility of the DeltaGrid approach to analyzing data dependencies among concurrently executing processes, but identified the centralized approach to data dependency analysis as a bottleneck in the process. The results presented in this thesis extend the data dependency analysis concept to decentralized approaches, where multiple Process Execution Agents maintain local delta object schedules and communicate as peers to share information about common data access patterns among concurrent processes.

CHAPTER IV

OVERVIEW OF PROCESS EXECUTION AGENTS

This chapter provides an overview of process execution agents that have been defined as part of this research (Urban et al., 2009b). The discussion begins in Section 4.1 with an example execution scenario that illustrates the construction of distributed process dependency graphs. Section 4.2 then describes the internal architecture of a PEXA as well as data structures created for the execution environment. Section 4.3 elaborates on the challenges associated with constructing distributed process dependency graphs.

4.1 PEXA Execution Scenario

The example execution scenario in Figure 4 assumes there are three PEXAs in the decentralized environment. Each PEXA is indicated as a rectangular box and is associated with a distributed site (D_i) that has a DEGS interface and possible multiple databases. Executing processes are indicated as circles, with lightning bolts indicating the PEXA that is controlling the execution of the process. A solid line from a process to a DEGS interface represents a service invocation. Dashed lines between PEXAs indicate decentralized communication among PEXAs. Data changes that are made by each DEGS are forwarded to the PEXA that is associated with the DEGS and stored in the local delta object schedule.

As shown in Figure 4, each PEXA is responsible for controlling the execution of local processes that are composed of service executions. Each process is invoking services that modify data at distributed sites. For example, site D_1 is controlling the execution of p_1 and p_4. Process p_1 is composed of two service executions identified as op_{11} and op_{12}, both executing at D_1. Process p_4 executes op_{41}, also at site D_1. Site D_2 controls the execution of p_2, where p_2 executes op_{21} at D_1 and op_{22} at D_2. Site D_3 controls the execution of p_3, which is executing op_{31} at D_2, op_{32} at D_1, and op_{33} at D_3.

As indicated in Figure 4, each invocation of an op_{ij} has a timestamp, t_x, indicating the time at which the operation is invoked. The box inside each PEXA provides a snapshot of the local delta object schedule for the data items that are being modified by each service that accesses data at the site, illustrating the interleaved data access by the service invocations of concurrent processes. For example, the delta object schedule for D_1 shows that objects X1 and Y1 have been modified. The schedule indicates the operations that have made the modifications and orders the schedule by the operation timestamps. The local schedule at D_1 indicates that p_2 is dependent on p_1 since op_{21} has modified X1 after op_{11} has modified X1 and p_1 is still executing. The schedule also indicates that p_4 is dependent on p_3 through access to Y1. At D_2, the operations have accessed data item X2, with the local schedule indicating that p_3 is dependent on p_2.

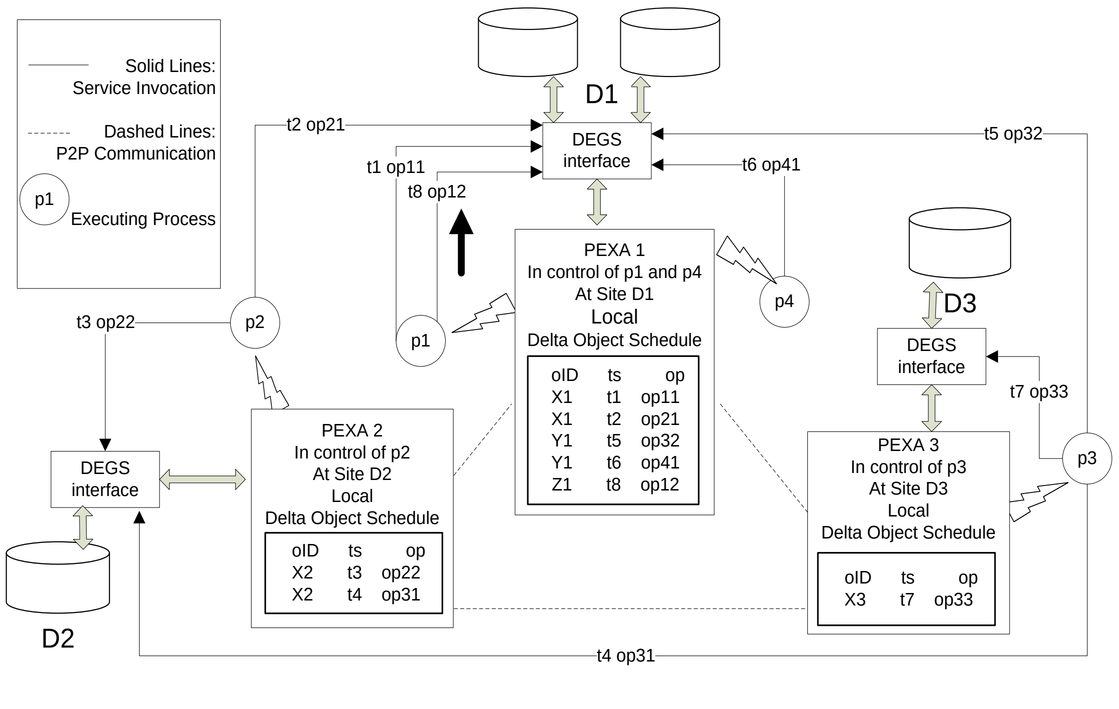


Figure 4: PEXA Execution Environment

4.2 Internal PEXA Architecture

Figure 5 shows the internal architecture of a PEXA. A PEXA contains a process execution component, such as a BPEL processor, with a Process History Capture System that records runtime information about the status of each executing process. Our implementation uses the db4o object-oriented database (Paterson et al., 2006) to record the runtime status of each process and to record the data changes that are communicated to the PEXA from each DEGS associated with the PEXAs local environment.

The local delta object schedule is the indexing structure showed in Figure 2 that sequences data changes in the delta repository according to time stamps and allows the recovery system to 1) analyze data dependencies and 2) retrieve delta information at different levels of granularity (e.g., all changes associated with a specific process or all changes associated with a specific service invocation within a process). The data dependencies are used by the recovery algorithm to identify processes that are write dependent on a failed process. There is no explicit data about read dependencies, so potential read dependencies are identified using runtime information about overlapping service execution as defined in (Xiao, 2006; Xiao and Urban, 2008b). Dependent processes can then query delta values, checking user-defined conditions to determine if they need to recover (i.e., execute compensating procedures) or continue running.

As part of the recovery process, a PEXA builds a process dependency graph based on the information in its local delta object schedule. But since a process can execute services at multiple sites, each monitored by a different PEXA, a PEXA must communicate with other PEXAs to construct a global, distributed view of process dependencies when a process fails. Furthermore, local process dependency graphs are extended with a structure known as a *link object* to assist in the construction of the global, distributed view. Section 4.3 elaborates on the use of link objects and other runtime information to construct global, distributed process dependency graphs.

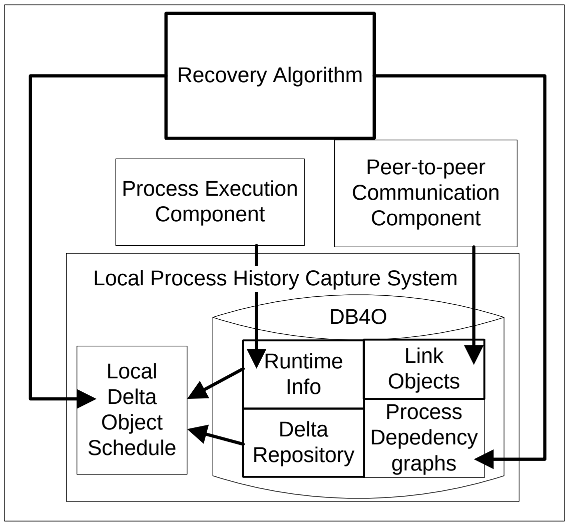

Recovery Algorithm
Process Execution Component
Peer-to-peer Communication Component
Local Process History Capture System
DB4O
Local Delta Object Schedule
Runtime Info
Link Objects
Delta Repository
Process Depedency graphs

Figure 5: Internal PEXA Architecture

4.3 Challenges for Decentralized Data Dependency Analysis

The objective of decentralized data dependency analysis is to construct a virtual, global process dependency graph to determine all active processes that are potentially affected by the recovery of a failed process. For example, if p_2 is dependent on p_1 and p_3 is dependent on p_2, then if p_1 fails, the global process dependency graph is $p_1 \leftarrow p_2 \leftarrow p_3$. As a simplification, this research assumes that a failed process and every dependent process of the failed process executes a compensating procedure as part of the recovery process, creating a cascaded recovery process. This is a worst-case scenario for constructing the full process dependency graph. Extensions to this simplification are addressed at the end of this thesis in the context of future research directions for the use of user-defined correctness conditions.

If the data changes for all active processes are in one delta object schedule, as in (Xiao, 2006; Xiao and Urban, 2008b), the construction of a global process dependency graph is straightforward. The challenge with the use of multiple PEXAs is that the delta object schedule is distributed among several PEXAs. As a result, a global view of process dependencies must be discovered through PEXA communication.

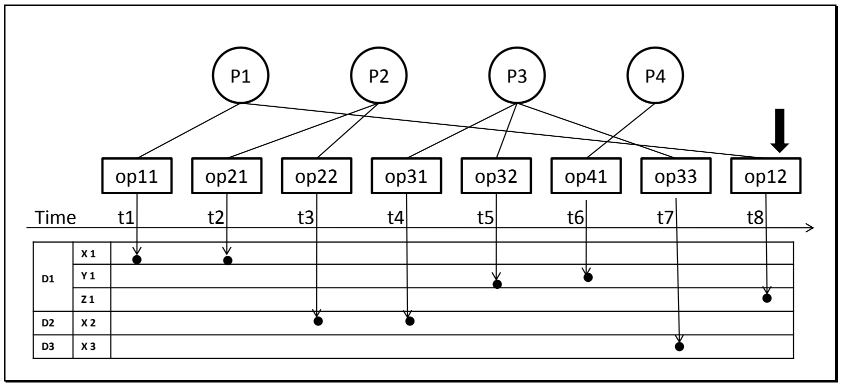


Figure 6: Data Access View of Interleaved Execution

As an example, consider again the process execution scenario in Figure 4. Figure 6 shows the interleaved execution view of each process and operation from a data access point of view when op_{12} fails at time t_8. The global process dependency graph for the four active processes is shown in the upper right of Figure 7, indicating that the process dependency graph is $p_1 \leftarrow p_2 \leftarrow p_3 \leftarrow p_4$. The recovery process is invoked when op_{12} fails at site D_1 and invokes the compensation of p_1, which is controlled by PEXA 1. Figures 6 and 7 together illustrate that PEXA 1 can detect that p_2 is dependent on p_1 due to modification of X1. PEXA 1 can also detect that p_4 is dependent on p_3 due to modification of Y1, but PEXA 1 cannot identify this dependency as part of the global graph for p_1 because of the distributed nature of the execution. As shown in Figure 7, p_3 is not dependent on p_1, p_2, or p_4 based on data access patterns at D_1, but p_3 is dependent on p_2 based on data accessed at D_2. Disconnected graphs such as those in PEXA 1 of Figure 7 are referenced to as *hidden dependencies*. Additional execution information must be recorded to link together all distributed components of the graph and to identify hidden dependencies within a single PEXA.

In particular, the runtime information about processes must be extended to record information about the distributed execution. When a service is executing at a PEXA, it is important to record whether the service is invoked by an internal or an external process. An internal process is a process that is controlled by the PEXA where the service is invoked. An external process is a process that is controlled by a PEXA different from the one where the service is invoked. For example, in Figure 6, op_{21} executes at the site of PEXA 1 but is invoked by a process running at PEXA 2. As a result, p_2 is marked as an external process (EX) in PEXA 1 within Figure 7. Using the same rationale, p_3 is marked as external in PEXA 2 (because of op_{31}) and also in PEXA 1 (because of op_{32}).

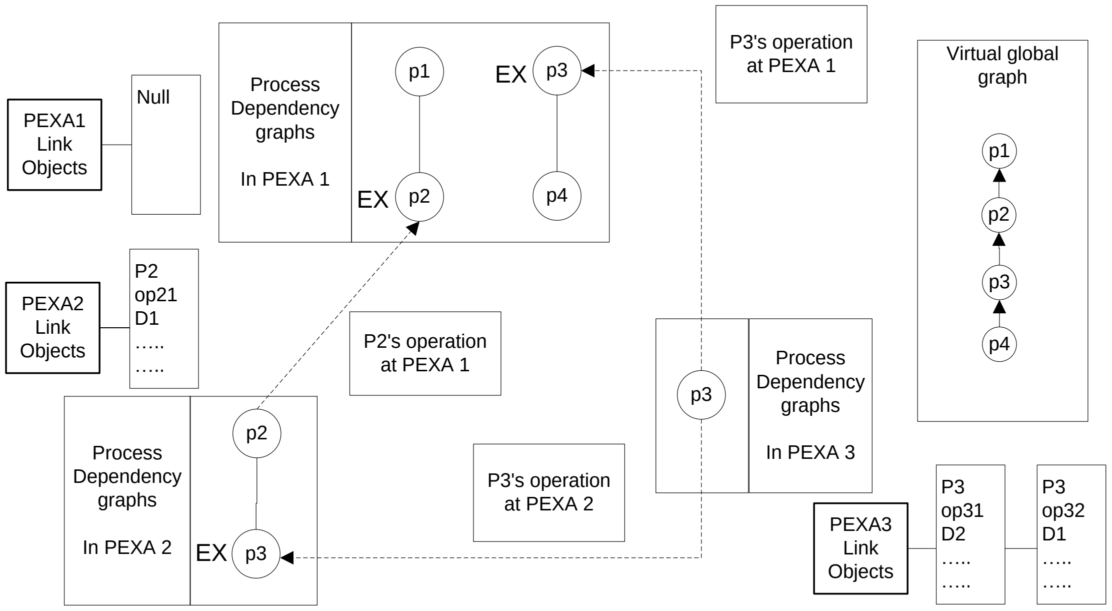


Figure 7: Global, Distributed Process Dependency Graph

In the opposite direction, a PEXA that controls a process that invokes a service at a different site must create a link object to record information about the site where the service is executed. In Figure 7, PEXA 2 creates a link object to indicate that op_{21} of process p_2 is executed at the site of PEXA 1. PEXA 3 creates two link objects to record the fact that op_{31} executes at PEXA 2 and op_{32} executes at PEXA 1. Used in combination, link objects together with an indication of internal or external process invocation can be used to dynamically discover global, distributed process dependency graphs. Section 5 elaborates on the algorithm for constructing distributed process dependency graphs among decentralized PEXAs.

CHAPTER V

DECENTRALIZED DATA DEPENDENCY ANALYSIS

Two recovery algorithms are proposed to achieve decentralized data dependency analysis. The lazy algorithm assumes that every process runs successfully and that a PEXA does not start to build the process dependency graph until one process fails. The eager algorithm dynamically builds graphs at runtime during service. As a result, process dependency graphs are available as soon as any process fails. Section 5.1 addresses the lazy algorithm (Urban et al., 2009b), describing dependency graph construction under the lazy approach and propagation of the recovery process among multiple PEXAs. Section 5.2 addresses the eager algorithm, describing dependency graph construction during process execution. Both algorithms are demonstrated using the execution scenario from Figure 4.

5.1 Dependency Analysis Using the Lazy Approach

The distributed graph construction and recovery algorithm is invoked upon the failure of a service within a process. The approach is to construct an initial process dependency graph at the site of the failure by calling findProcessDependencies(processId), where processId is the identifier of the failed process. The graph is then used to 1) recover local service executions and 2) find information about external processes and link objects to communicate with other PEXAs about propagation of recovery and graph construction activities. Link objects point to services that are under the control of a process at the current PEXA but were executed at a different PEXA, whereas services marked as external (EX) have executed at the current PEXA but are under the control of a process at a different PEXA.

5.1.1 Preliminary Issues for Graph Construction and Analysis

The process dependency graph data structure is created to store information about data dependencies at the process level. Let op_{jk} represent a service invoked from process p_j and op_{mn} represent a service invoked from process p_m. If op_{mn} is write dependent (or potentially read dependent) on op_{jk}, then p_m is identified as dependent on p_j in a process dependency graph for p_j when p_j fails. In the graph, nodes represent processes and edges represent process dependencies. For example, if p_1 is dependent on p_2, then the dependency $p_2 \leftarrow p_1$ is stored in the graph as two nodes, with an edge pointing from p_1 to p_2. The graph is represented as a hashmap called adjacencyMap that combines a key-value pair for fast retrieval, where a process is a key and its value is a list to store all processes that are immediately read and/or write dependent on another process. Dependencies are found using procedures in (Xiao, 2006) for querying a delta object schedule. After finding immediate dependencies, transitive dependencies are recursively found.

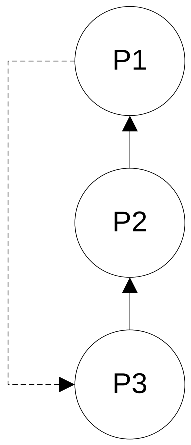


Figure 8: Cycle in the graph constructed by the lazy approach

There can potentially be cycles in a process dependency graph. For example, in Figure 8, suppose the following cycle exists: $p_1 \leftarrow p_2 \leftarrow p_3 \leftarrow p_1$ when p_1 fails, where p_1 and p_3 are dependent on each other. The dependency of p_3 on p_1 was created before the dependency of p_1 on p_3. For the lazy algorithm, since the graph is constructed to control the order of the recovery process, a cycle when detected is not needed in the graph. In the above example, p_1 will be recovered before p_2 and p_2 will be recovered before p_3. As a result, it is not necessary to enter the cycle in the graph since p_1 is recovered already before p_3. Here, compensation is used for recovery, which means to logical undo the previous results and return to pre-defined results. The difficulty with cycles is that the graph is distributed. A PEXA must therefore be capable of dealing with local and global cycles.

Local cycles can be detected using information in the local delta object schedule. The method addVertex(p_i) in Figure 9 is used to add nodes that represent processes (p_i) to the graph (g). A process is added to a graph only if a node representing the process does not already exist. The method addEdge(p_i, p_j) in Figure 10 is used to create an edge in g, indicating that p_j is dependent on p_i. To avoid local cycles, the method addEdge(p_i, p_j) prevents cycles by first checking to see if p_j is already a parent of p_i in the graph. If so, the edge is not created to avoid a cycle. The variable result is a list variable to share the current value of the dependency graph using a breadth first traversal. The traversal() method in Figure 11 is used to do a breadth first traversal of the graph and return the value of result.

```
public boolean addVertex (String vertex)
{
    if (adjacencyMap.containsKey(vertex))
        return false;
    adjacencyMap.put (vertex, new LinkedList());
    return true;
}
```

Figure 9: addVertex() Procedure

```
public boolean addEdge (String v1, String v2)
{
   LinkedList l = (LinkedList)traversal(v2);
   LinkedList ll = (LinkedList)adjacencyMap.get(v1);

   if(!l.contains(v1)){
      ll.add(v2);
      System.out.println("add an edge!");
   }
   else
      System.out.println("Cycle eliminated!");
   return true;
}
```

Figure 10: addEge() Procedure

```
public List traversal(String root){
    if(adjacencyMap==null) return null;
    if(!result.contains(root)) result.add(root);
    List temp=new LinkedList();
   temp=(LinkedList)adjacencyMap.get(root);
   if(temp==null)  return null;
    for(int i=0;i<temp.size();i++)
    {
       if(!result.contains(temp.get(i))) result.add(temp.get(i));
    }
    for(int j=0;j<temp.size();j++)
       traversal((String)temp.get(j));
    return result;
}
```

Figure 11: traversal() Procedure

Information about a service execution that was requested by an external process is stored in the runtime information component of a PEXA. The structure of an entry in the schedule is:

- pName (the process name)
- pId (the process identifier)
- opName (the operation name)
- opId (the operation identifier)
- oId (the object identifier)
- PEXAId (the controlling PEXA)
- inOrEX (indicating whether a process is local or external)
- status (the execution status of the process)

The inOrEx field distinguishes between service execution requested by a local (i.e., internal) process and service execution requested by an external process running at another PEXA. This information is queried during the graph construction process to indicate that notifications must be sent to the corresponding PEXA about propagation of the recovery and graph construction process.

Because the processes in the process dependency graph can come from multiple PEXAs as remote operations composing processes, the location of a process in the controlling PEXA needs to be identified so that the dependency analysis can be conducted in a decentralized manner. Link objects are used to support this capability. Link objects are virtual references to the external operations and are created by a PEXA when a process executing at a local PEXA invokes a service at a remote site. The structure of a link object is:

- processId (identifier of the controlling process)
- opName (name of the service)
- opId (service identifier)
- degsId (DEGS identifier)
- status (indicating successful or compensated)

A db4o database is used to store the link objects of each PEXA. During recovery, a PEXA can compensate all its operations, both local and remote ones. The use of link objects supports the detection of the hidden dependencies. Link objects are also needed for propagation of the recovery and graph construction process. The link object attribute status is used to address distributed cycles. The attribute indicates the status of an external operation as either successful or compensated. When an external operation finishes executing successfully, it will send its successful status back to the controlling process and update the corresponding link object. If the service is later compensated at the execution site, a notification will be sent back to the controlling process to change its status to compensated. This value is used in the propagation of the recovery and graph construction process to avoid distributed cycles (i.e., to prevent invoking compensation of procedures that have already been compensated). The use of this value and the decentralized algorithms will be illustrated in the following two sections.

5.1.2 The Lazy Algorithm

Figure 12 provides pseudocode of the graph propagation for the lazy algorithm. This procedure is called after the PEXA finds out that a process fails and the failed process is passed to the procedure findProcessDependencies(processId). Two list variables for dependency detection are created, processWDon for write dependency and processRDon for read dependency. These lists indicate the processes on which the service is write or potentially read dependent.

The findProcessDependencies() procedure first finds all of the immediate dependent processes of the failed process, both write dependencies and potential read dependencies. Write dependencies are collected from the local delta object schedule by the method getWriteDependentProcessListOnProcess(processId) and returned to processWDon. The potential read dependencies are from the runtime information by using the method getReadDependentProcessListOnProcess(processId) and returned to processRDon. In (Xiao, 2006), every concurrent process is suspended to execute recovery procedures and resumes after the recovery. In this research, these two methods contain procedures to lock the data items of dependent processes. So if there are concurrent processes trying to access locked data items, they have to wait for the release of locks held by the recovery processes. Processes accessing other data items continue running. Compared with suspending everything in the previous work, the locking of data items is more efficient and reasonable. Each of the two methods will return lists of processes dependent on the failed process. Since there could be duplicate processes in these two list, they are merged into one list and sent to the recursive graph construction method buildGraph(list, processed, graph, n), where list is the merged dependent process list and n is a value to make sure that a node from the list is considered only once. As shown in Figure 13, the buildGraph() method is invoked to run through each of the dependent processes, creating nodes and edges in the local process dependency graph.

After graph construction, the traversal() procedure is called to do a breadth first traversal of the graph and generate an ordered list of processes that is returned for use in the recovery process.

Figure 14 provides pseudocode for the recovery process. The procedure is called after the construction of the local process dependency graph and is passed the ordered list of processes to be recovered.

```
public void findProcessDependencies (String processId)     //failed process id
 {
      //create a new vector
      Vector pListWD=new Vector();
      Vector pListRD=new Vector();

      //create a new list to store all the dependent  processes based on the failed one
      List result=new LinkedList();

     // n is used for building graphs
     int n=0;

      //get all the processes that are write dependent on failed process or read
      //dependency
      pListRD=ProcessInfoAccess.getReadDependentProcessListOnProcess(processId)
      pListWD=GlobalScheduleAccess.getWriteDependentProcessListOnProcess(processId);

      //merging lists procedure eliminates duplicated processes
      Vector newList=merge(pListRD, pListWD);

      //building graphs
      Graph g=new Graph(processId);

      //recursively iterate through every dependent process
      buildGraph(newList, processId, g, n);

      //breadth first traversal
      result=g.traversal( processId );

     //start the recovery process for the graph
      recover(result);

    }
```

Figure 12: findProcessDependencies() Procedure

The recover() procedure examines each process in the list and determines if each process is internal or external. All local service executions are identified and recovered. Recall that we are initially assuming that every process is recovered through compensation of each service invocation. Since each internal process can also invoke services at other sites, the algorithm then queries the link objects associated with the process to find services of the process that were executed at other sites (i.e.,

```
// recursive method to build dependent processes
public void buildGraph(Vector pList, String processId, Graph g, int n)
{
        //temporary value temp1 to pass processId to
        String temp1=processId;

        //whether there are dependent processes
        if(pList.size()!=0)
        {
            //start to build graph by adding the vertex
            g.addVertex(processId);

            //check each of the dependent processes
            for(int i=0;i<pList.size();i++)
            {
                 //use a temporal variable
                ProcessInfo temP=(ProcessInfo)pList.get(i);

                //add vertex
                g.addVertex(temP.getProcessId());

                 //add edge
                g.addEdge(temp1, temP.getProcessId());

                //get the process id
                temp1=temP.getProcessId();

                //find all the processes write and read dependent on temp1 (or read dependency)
                Vector temPListRD
                =ProcessInfoAccess.getReadDependentProcessListOnProcess(processId)

                Vector temPListWD
         =GlobalScheduleAccess.getWriteDependentProcessListOnProcess(temP.getProcessId());

                //merge two lists
                Vector newList=merge(tempListRD, tempListWD);

                //check the current process dependency and keep building the graph
                buildGraph(newList, temp1, g, n);
            }
        }end if
    }end for
  }
```

Figure 13: buildGraph() Procedure

```
// recover dependent processes according to where they come from
public void recover( List list){

   //create a new list for operations from the lcoal schedule
   List tempList;

   FOR each process in the list
   {
      //find operations from the lcoal schedule
      tempList = (List)ProcessInfoAccess.getExecutedOperationList(processId);

      // there are operations to be compensated
      if(tempList!= null){
         compensate(tempList);
      }

      IF the process is initiated by the local PEXA
      {
            //find external operations of processId from the link objects table
            tempList=LinkObject.getExecutedOperationList(processId);

            //send notifications
            if(tempList!=null)
                 sendNotification(tempList);
      }
      ELSE //the process is initiated by a peer PEXA
      {
            //send notifications when the process is not a root node in the graph
            if( ! g.isRoot(processId) )
                 sendNotification(processId);

      }

   }END FOR

}
```

Figure 14: recover() Procedure

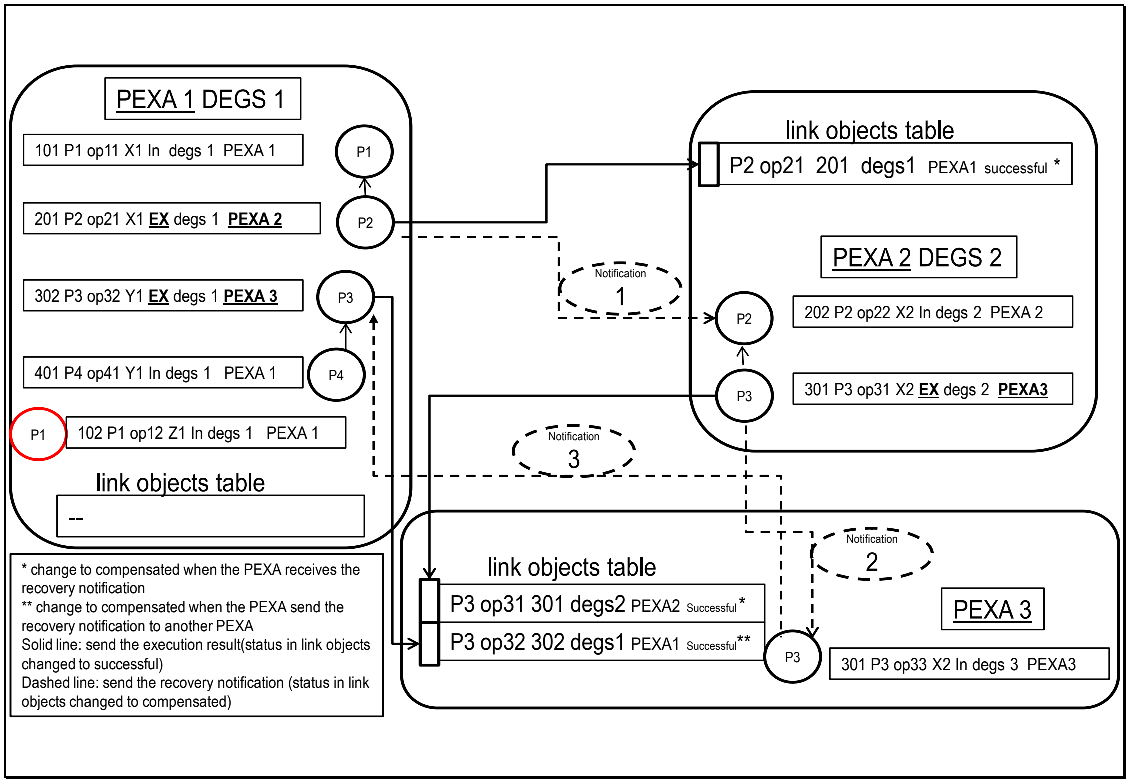
the IF part of the algorithm). Notifications are then sent to the PEXAs of each external process. Each PEXA will then invoke findProcessDependencies(processId) for the relevant process to construct its own local dependency graph to continue the recovery process at the new PEXA site.

For a service invoked by an external process, the service is compensated and then a notification is sent to the external PEXA to propagate the recovery and graph-building process. The notification includes information about changing the status of the corresponding link objects to compensated. Once a graph is compensated, it is deleted.

5.1.3 Execution Scenario for the Lazy Algorithm

Figure 15 uses the execution scenario from Figure 6 to illustrate the logic of the algorithm presented in Figures 12-14. When the execution of an external operation is completed, the execution result is sent back to its controlling PEXA to mark the status in its link object. This communication is shown as solid lines between PEXAs in Figure 15. Notifications that are initiated by the sendNotification() procedure are drawn as dashed lines in Figure 15.

In the scenario from Figure 6, the recovery process is initiated when op_{12} fails in PEXA 1 and constructs a local process dependency graph. Recall that link objects have already been created for each process as a result of execution up to this point. In PEXA 1, the local dependency graph is initially determined to be $p_1 \leftarrow p_2$. In Figure 15, the box to the left of each process node shows the runtime information for the process, indicating the service executed and the internal/external status of the associated process. The recover procedure for the graph compensates procedure op_{11}, which is an internal service. There are also no entries for p_1 in the link object table, indicating that all of p_1's services were executed at site D_1. As a result, tempList is null and no notifications are sent. Since p_2 is an external process, op_{21} is compensated at PEXA 1 and then a notification is sent to PEXA 2 (labeled as notification 1 in Figure 15), indicating that 1) op_{21} should be marked as compensated in the link object table and 2) the recovery and graph construction process should continue at PEXA 2 using p_2 as a root node (i.e., invoke findProcessDependencies(p_2)).


PEXA 1 DEGS 1
101 P1 op11 X1 In degs 1 PEXA 1
201 P2 op21 X1 EX degs 1 PEXA 2
302 P3 op32 Y1 EX degs 1 PEXA 3
401 P4 op41 Y1 In degs 1 PEXA 1
102 P1 op12 Z1 In degs 1 PEXA 1
P1
P2
P3
P4
P1
link objects table
--
link objects table
P2 op21 201 degs1 PEXA1 successful *
PEXA 2 DEGS 2
202 P2 op22 X2 In degs 2 PEXA 2
301 P3 op31 X2 EX degs 2 PEXA3
P2
P3
Notification 1
Notification 3
Notification 2
link objects table
P3 op31 301 degs2 PEXA2 Successful *
P3 op32 302 degs1 PEXA1 Successful**
PEXA 3
301 P3 op33 X2 In degs 3 PEXA3
P3

* change to compensated when the PEXA receives the recovery notification
** change to compensated when the PEXA send the recovery notification to another PEXA
Solid line: send the execution result(status in link objects changed to successful)
Dashed line: send the recovery notification (status in link objects changed to compensated)

Figure 15: Execution Scenario

At PEXA 2, the graph $p_2 \leftarrow p_3$ is created from the local delta object schedule. The algorithm in Figure 14 is then invoked to recover the operations associated with the graph. The first iteration through the recover procedure determines that p_2 is an internal procedure, finding a local operation (op_{22}) and a remotely executed operation (op_{21}). PEXA 2 will compensate op_{22} and discover that op_{21} has already been compensated. As indicated in the comment box in Figure 15, successful* is changed to compensated for op_{21} in the PEXA 2 link object table when notification 1 is received.

When p_3 is processed, it is identified as an external node. As a result, op_{31} is compensated and notification is sent to PEXA 3 (notification 2 in Figure 15) to propagate the recovery and graph construction process, together with information about changing the status of the link object for op_{31} from successful* to compensated.

At PEXA 3, the graph contains only one node for p_3, which in an internal process. When the algorithm in Figure 14 is invoked, the IF part of the code is then executed. As a result, op_{33} is compensated since it was executed at PEXA 3. Link objects are then found for op_{31} and op_{32}. Since op_{31} has already been marked as compensated, the notification message is only sent to PEXA 1 for the invocation of findProcessDependencies(p_3). The status of op_{32}'s link object is changed from successful** to compensated before sending the notification, with the actual compensation to take place at PEXA 1.

PEXA 1 constructs the graph $p_3 \leftarrow p_4$. Since p_3 is an external node, op_{32} is compensated at PEXA 1 and a notification is sent back to PEXA 3 (not shown in Figure 15). PEXA 3 will be able to determine at this point that all relevant services for p_3 have already been compensated and thus will not continue to propagate the process (i.e., detects and terminates a distributed cycle). PEXA 1 then compensates op_{41} and terminates since there are no more notifications to send.

Note that when the findProcessDependencies() procedure is called in each PEXA to construct a local process dependency graph, the data items identified in the local delta object schedule are locked, with compensating procedures executing as nested transactions that inherit the associated locks. This prevents other executing processes from accessing the data involved in the recovery process and creating further dependencies.

5.2 Dependency Analysis Using the Eager Algorithm

Unlike the lazy algorithm, the eager algorithm dynamically builds the process dependency graph at runtime. As a result, whenever a service is invoked, the PEXA builds a graph using both its runtime information and deltas. As in the lazy algorithm, the graph is used to recover local service executions, using information about external processes and link objects to communicate with other PEXAs for recovery and graph construction.

5.2.1 The Eager Algorithm

Figure 16 illustrates the difference between the lazy and eager algorithms. As mentioned in Section 5.1, the lazy algorithm is invoked on the failure of an operation from a process. To determine data dependencies, the algorithm reads forward in the delta object schedule to discover processes dependent on the failed process.

The eager algorithm detects dependencies dynamically at runtime instead of waiting for the failure to occur and then builds process dependency graphs during execution. When a process fails, the dependent processes are ready to be recovered since the process dependency graph is dynamically maintained. When an operation of a process completes, the object schedule is scanned backwards in time to determine processes on which the completed process is dependent. If dependencies are identified, the process dependency graph will be updated. If not, a new graph with a single root node for the process of the completed operation is created.

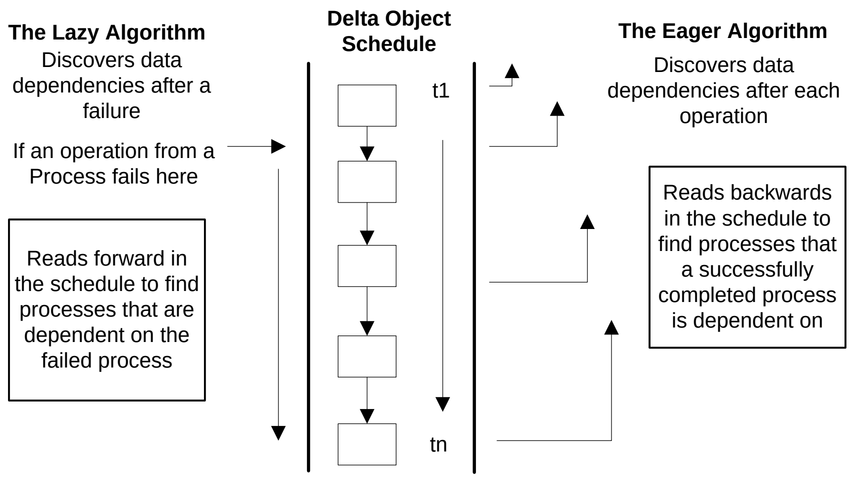

The Lazy Algorithm
Discovers data dependencies after a failure
If an operation from a Process fails here
Reads forward in the schedule to find processes that are dependent on the failed process
Delta Object Schedule
t1
tn
The Eager Algorithm
Discovers data dependencies after each operation
Reads backwards in the schedule to find processes that a successfully completed process is dependent on

Figure 16: Difference between Lazy and Eager Algorithm

The link object information, which represents external service executions, is still recorded for recovery consideration. The advantage of the eager approach is that when an operation fails or a notification triggers the recovery process, PEXAs already have the dependent processes and are able to initiate the recovery process immediately. The eager algorithm, however, has overhead associated with process dependency graph construction for every process.

Figure 17 provides pseudocode of the graph propagation for the eager algorithm, findProcessDependencies(). When the graph construction procedure is invoked after the completion of each operation, write and read dependencies are discovered and process dependency graphs are updated accordingly with new elements, such as a new edge or a new separate node.

To generate the processWDon list, deltas created by the current process are examined to get the data items that have been modified. Since there can be more than one data item that has been modified, the data items are recorded into processWDon by identifiers. A procedure is also called to get the potential read dependencies, which also returns the read dependencies to the processRDon list. After merging the two lists to avoid duplicates, edges are added in to the graph pointing from the current process to its dependent processes.

The graph is built at the process level under the eager algorithm. A completed process only needs to record its most immediate dependencies. For example, suppose, p_1, p_2 and p_3 have all modified data item X in the order of p_1 at t_1, p_2 at t_2, and p_1 at t_3. When p_3 completes, it records that it is dependent on p_2. p_2 will record the dependency on p_1, creating the transitive dependency of p_3 on p_1. As a result, it is not necessary to explicitly record the dependency of p_3 on p_1.

If processId does not exist in the current graph, a new vertex is added. The processId and operationId of the completed process is then passed into the checkLastModificationOnSameDataItem(processId, operationId) procedure of Figure 18 to find the data items that have been modified. During process execution, when a data item is modified by an operation from a process, processId and objectId are recorded separately as a key and value in latestOperationOnData. As a result, when the modified data items of a completed process are identified, the corresponding latest process can be discovered as well. The checkLastModificationOnSameDataItem() procedure returns a list containing all of the latest processes on which the completed process is dependent according to each data item that was modified.

```
public void findProcessDependencies(String processId, String operationId){

   //a list to store all the operations that this operation is write or read dependent on
   boolean nodeInGraph=checkExisting(processId);
      Vector processWDon=new Vector();
      Vector processRDon =new Vector();
      Vector merge=new Vector();
      //if it's not existing, add node
      if(!nodeInGraph){
         g.addVertex(processId);
      }
         processWDon =CheckLastModificationOnSameDataItem(processId, operationId);
         processRDon
         =ProcessInfoAccess.getReadDependentProcessListOnProcess(processId);
         merge=merge(processWDon, processRDon);
         //if there is a process that the current process is dependent on
         if(merge!=null){
            //add edge a ---> b for each dependent process
            for(int i=0;i<merge.size();i++)
               g.addEdge((String)merge.get(i), processId);
         }
}
```

Figure 17: Graph Propagation for the Eager Algorithm

```
public Vector checkLastModificationOnSameDataItem(String processId, String operationId){
      Vector result=new Vector();
      Vector dlist=GlobalScheduleAccess.getDeltas(processId, operationId);
      if(dlist==null)  return null;
      for(int i=0;i<dlist.size();i++){
         Delta temp=(Delta)dlist.get(i);
         String dataItem=temp.getObjectId();
         String latestProcess= (String)Server.latestOperationOnData.get(dataItem);
         result.add(latestProcess);
      }
      return result;
   }
```

Figure 18: checkLastModificationOnSameDataItem() Procedure

The eager approach assumes that a process will potentially fail and collects all the dependencies and builds graphs for fast recovery. Therefore, when a process actually fails, the relevant dependent processes already exist and are recovered using the same recover() procedure in Figure 14 as in lazy algorithm. If a process fails, the graph will be retrieved by the traversal() procedure passing the identifier of the failed process as the parameter to recover the sub-graph that represents the dependent process list.

5.2.2 Execution Scenario for the Eager Algorithm

The example in Figure 19 shows how to build a process dependency graph using the eager algorithm. The left side of the figure shows the order of execution for the operation of processes p_1, p_2, and p_3. The right side of the figure shows the dependency graph that is constructed along with the process execution at runtime.

In Figure 19, op_{11} of p_1 executes and modifies data item object1. After its execution, the findProcessDependencies() procedure is invoked to decide whether p_1 is dependent on other processes. At this point, p_1 is not dependent on any previous processes. As a result, a node will be added in a new graph for p_1.

After the execution op_{21} of p_2, based on the data item it has modified, p_2 is not dependent on the other processes. So a node for p_2 is also added to the graph. When op_{22} of p_2 executes, the findProcessDependencies() procedure discovers that the latest process operating on object1 is op_{11} from p_1. Hence an edge pointing from p_2 to p_1 is created in the graph to represent the dependency labeled as Edge1 in Figure 19.

After op_{12} of p_1 executes, the dependency of p_1 on p_2 is discovered. This edge, indicated as Edge2 in the graph of Figure 19, is also added, creating a cycle in the graph. Unlike the lazy algorithm, cycles are allowed in the graph since the order of dependent processes to be compensated is volatile according to different failed processes. Cycles, deletion of nodes, and other process dependency graph issues are addressed in Section 5.2.4.

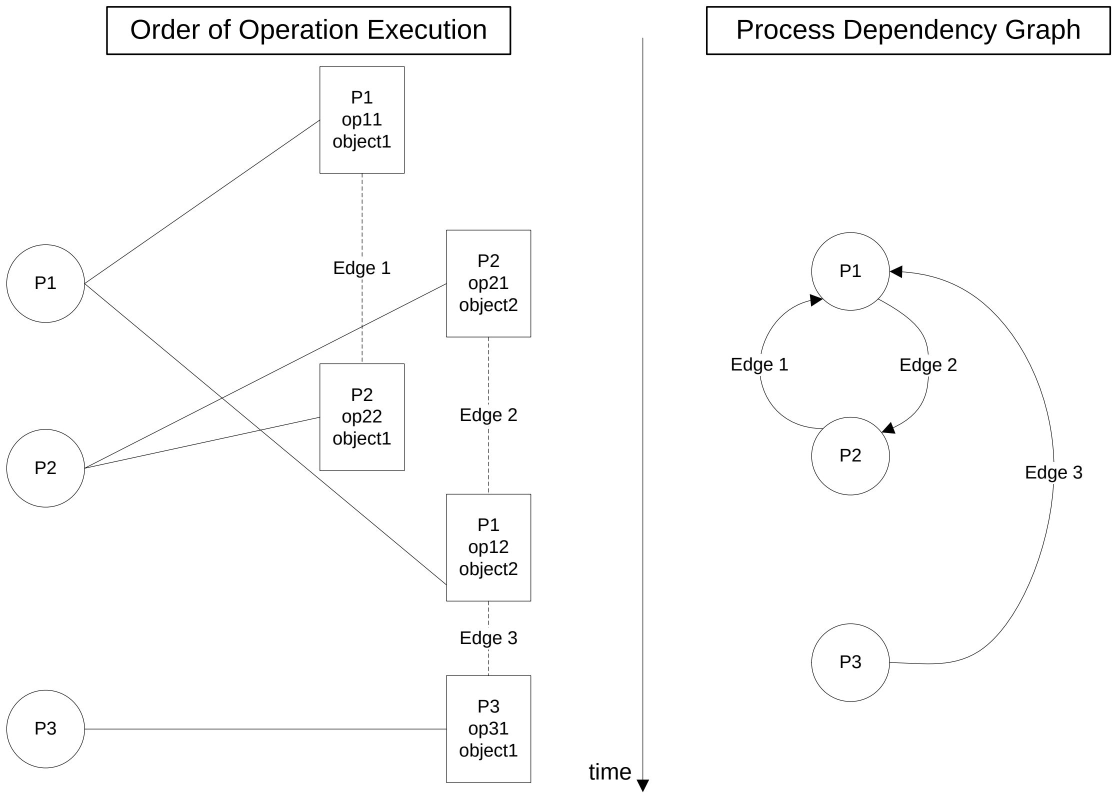


Figure 19: Process Dependency Graph Construction with the Eager Approach

After op_{31} of p_3 executes, p_3 is dependent on both p_1 and p_2 according to data item object2 that they have modified. However, p_1 has the latest modification to the data item that p_3 has modified. So the dependency of p_3 on p_1 is added to the graph, as Edge3 in Figure 19.

5.2.3 Decentralized Scenario for the Eager Algorithm

Figure 20 uses the distributed execution scenario from Figure 15, to illustrate the use of the eager algorithm with a decentralized dependency graph. Processes execute concurrently in three PEXAs and graphs are constructed at runtime. At the end of an operation execution, the findProcessDependencies() procedure is invoked to update the process dependency graph data structure, either to generate a root node or to add a node and edges to an existing graph.

In PEXA 1, after op_{11} of p_1 executes on X_1, the application calls the graph construction procedure. The deltas created by this operation are retrieved to find the modified data items for the write dependencies to add in the processWDon list. Here, data item X1 is found modified by p_1. For each data item found by delta retrieval, only the latest operation on the item is needed to construct the graph if there is any. The HashMap structure latestOperationOnData is used to retrieve the latest operation corresponding to each data item since this variable records the pair of the latest process and data item. At this time, there is no process that has modified X_1. As a result, no dependencies are discovered and only a node for p_1 is added to the graph structure.

After the execution of op_{21} of p_2 from PEXA 2, the findProcessDependencies() procedure is invoked. The deltas that op_{21} has created are retrieved. The result returns X1. Based on X1, write dependencies are analyzed and p_1 is found to be the latest operation to modify X1. p_1 is added in the processWDon list. Then the node p_2 is added in the graph and an edge from p_2 to p_1 is also created.

Meanwhile, op_{22} of p_2 is executing in PEXA 2 modifying data item X_2. Since no dependency exists at this point, the graph in PEXA 2 is created with p_2 as a root node. After p_3 executes op_{31} in PEXA 2, the findProcessDependencies() procedure is invoked. The delta created by p_3 indicates the data item modified is X_2 and, according to the latestOperationOnData variable, p_2 is the latest operation that has modified the same data item before p_3. Therefore, p_3 is added in the graph as a node and an edge from p_3 pointing to p_2 is also added for the dependency discovered.

After p_3 creates a delta in PEXA 1, the findProcessDependencies procedure is invoked. The delta indicates data item Y1 has been modified. Since no other process has been found to have modified this data item, p_3 only generates a node in the graph with no edges. Then p_4 executes and generates a delta by modifying Y1. The latest process that has modified Y1 is p_3, resulting in an edge from p_4 pointing to p_3 in the graph.

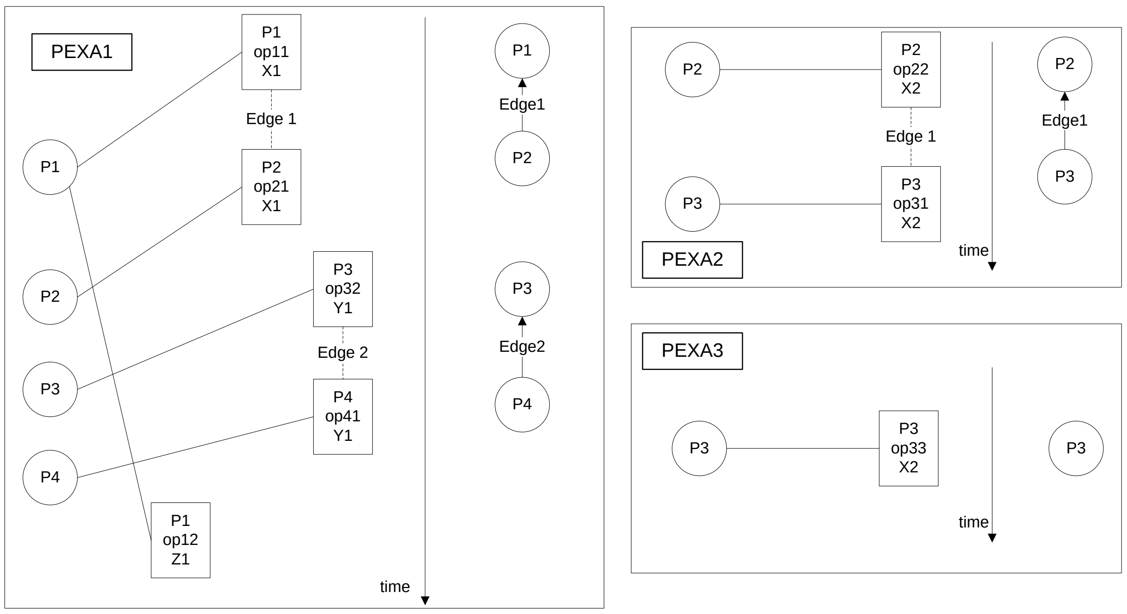


Figure 20: Decentralized Execution Scenario for the Eager Algorithm

In PEXA 3, after op_{32} of p_3 invokes a service in PEXA 2, op_{33} of p_3 executes locally and creates a delta by modifying a data item X3. Since no other process can be found to have modified X3, a graph is generated with only one node p_3 as a root. Then, p_3 remotely invokes a service at PEXA 1.

At this point, there are two graphs in PEXA 1, one graph in PEXA 2 and one in PEXA 3. When op_{12} from p_1 executes locally at PEXA 1 and fails, the same recover() procedure in Figure 14 used by the lazy algorithm is invoked. Since the process dependency graphs exist already, the sub-graph based on the problematic process can be simply retrieved. The rest of the recovery is the same with that of the lazy algorithm.

5.2.4 Process Dependency Graph Issues

In this section, maintenance issues are addressed for the use of the eager approach. Section 5.2.4.1 discusses the issue of cycles. Section 5.2.4.2 discusses the deletion of nodes from the graph.

5.2.4.1 Cycles in the Process Dependency Graph

As illustrated in the previous sections, the eager approach allows cycles to appear in local process dependency graphs. It is necessary to represent cycles since the graph is constructed for all executed processes in anticipation of a possible failure. In comparison, the lazy algorithm only constructs a process dependency graph when a process fails. The graph for the lazy approach defines the order for recovery of dependent processes. As a result, dependency cycles are not relevant.

Using the eager approach, when a failure occurs, the sub-graph to be recovered is extracted from the graph by doing a breadth first traversal starting from the node that represents the failed process with the traversal() procedure, detecting and eliminating cycles for recovery. Using Figure 19 as an example, when P_1 in the graph fails, the sub-graph based on P_1 is $P_2 \rightarrow P_1 \leftarrow P_3$. When P_2 fails, the sub-graph based on P_2 is $P_2 \leftarrow P_1 \leftarrow P_3$. Since a failure can potentially happen to any active processes, different sub-graphs will be generated for different failed processes. And it is unlikely to predict ahead of time when a process will fail. Therefore, when a failure occurs, the graph is traversed to create the recovery order of dependent processes. Hence, cycles are needed in the graph construction for the eager algorithm.

5.2.4.2 Deletion of Nodes

Another important maintenance issue is to delete nodes from process dependency graphs, since the graphs will keep growing large with continuous executing processes. There are two situations to consider for the deletion of nodes. Examples are given in Figure 21. One situation occurs after the execution of the recovery procedure when sub-graphs need to be deleted from the graph structure.

Another situation requires that PEXAs using the eager algorithm periodically examine their own local graphs for the deletion of completed processes.

First consider the recovery situation. After recovery, all of the nodes representing processes in the sub-graphs need to be deleted from the local process dependency graphs to reduce the complexity of local graphs. Since the sub-graph contains all of the processes dependent on the failed process, deleting the sub-graph will not impact the other dependencies stored in the graph. Therefore, the processes in the sub-graphs can be removed without any problems. For example, in Figure 21, Situation 1 illustrates that Process i is dependent on Process g and Process h, which are dependent on Process f, and Process f is dependent on Process d. If Process f fails, nodes f, g, h and i can be removed from the graph after the recovery of the sub-graph based on Process f.

For the maintenance situation, process dependency graphs need to be checked periodically to remove processes that have successfully completed. Even if there are processes dependent on the completed processes or the completed processes are dependent on others, the completed processes can be removed since they have committed successfully and cannot be recovered. This research prescribes a top down rule for removal of completed processes, where completed nodes are deleted beginning from a completed root node.

If the completed process is a single root node in the graph, it can be removed from the graph. As a result, dependent processes, if there are any, become new single roots of their own graphs and are recursively examined to delete if they are dependent completed processes. For example, Situation 2.1 in Figure 21 indicates that Process a can be removed from the graph if Process a completes. Process j then becomes a root node and can also be deleted if it has completed, as indicated in Situation 2.2a. In some case, when a node such as Process a is deleted, a dependent node will be discovered to be part of a cycle. For example, after deleting Process a, Process b is identified as a dependent node but b is not a root node. Instead, Process b is part of a cycle since it cannot be removed if it has completed.

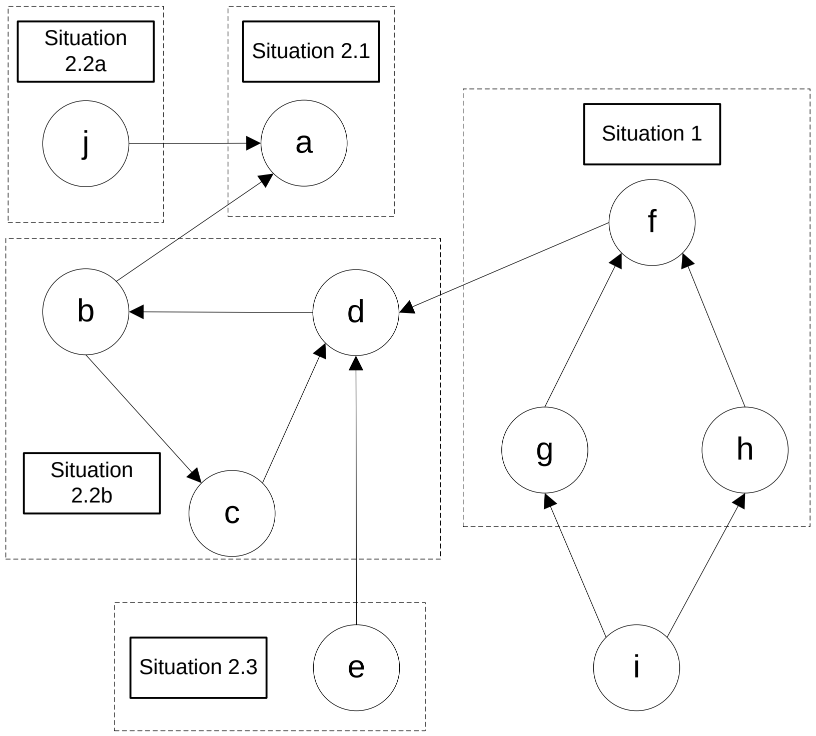


Figure 21: The Deletion of Nodes from the Graph

Cycles can be detected by some of the existing algorithms in the area of graph theory. Classic algorithms used to find cycles in a graph are the work of (Floyd, 1967) and (Brent, 1980).

After cycle detection, the only way to delete the cycle as a root node is to make sure that 1) every node in the cycle is a completed process; 2) none of the nodes is dependent on other nodes. The processes that are dependent on the processes in the cycle do not impact the deletion of the cycle. In the example of Situation 2.2 from Figure 21, to delete the cycle as a root, Process b, a and d cannot be dependent on other processes and have to be completed processes after Process a is deleted.

If a process is not a root node but has completed, it cannot be removed immediately to avoid complex additional re-structuring of the graphs. Therefore, completed processes which are not root nodes are marked as completed and wait to be removed when they become root nodes. During the retrieval of sub-graph where a failure occurs, nodes marked completed are not added in the sub-graph of nodes to be recovered. The processes that are dependent on successfully completed processes are not added to the sub-graph unless they have direct dependencies on the processes to be recovered. After the recovery of the sub-graph, the completed process is removed as part of regular maintenance since it is new root node now.

For example, in Situation 2.3 in Figure 21, Process e is dependent on Process d. So when Process e has completed, it is not removed until it becomes a root node. As another example using Situation 1, suppose that Process g has completed and then Process f fails. The sub-graph starts from Process f and then to Process g. Since Process g is completed, Process g is not added to the sub-graph to be recovered. Then Process h is added to the sub-graph followed by Process i is added. Therefore, the sub-graph is Process f, Process h, and Process i. Process g then becomes a root node and is removed as part of regular maintenance.

The maintenance of the process dependency graph guarantees that the local graphs are consistent and at minimal redundancy with the maximum effect of reducing the graph size for PEXAs.

CHAPTER VI

IMPLEMENTATION AND EVALUATION OF DECENTRALIZED DATA DEPENDENCY ANALYSIS

This chapter presents an analysis of the decentralized data dependency analysis algorithms. Section 6.1 describes the implementation environment and measurement criteria. Section 6.2 presents an evaluation of the recovery propagation algorithm that is central to the lazy and easy algorithm.

6.1 Implementation Environment and Measurement Criteria

The implementation was done in a Windows machine. The operating system used was Windows XP Professional x64 Edition with an Intel processor Core 2 Extreme Q6850 @ 3 GHz 4 GB of memory. Java was used to develop the PEXA architecture and distributed algorithms, as well as the delta generator using Netbeans 6.5 as the integrated development environment. The communication between PEXAs was set up using Java Sockets. Each PEXA has a socket server and client to send and receive messages from other PEXAs, for compensation or updating process status. In this experiment, three PEXAs were deployed

PEXAs can be deployed in several machines or in one machine by specifying either IP addresses of different machines or ports of the same machine. This research mainly focused on analyzing different aspects of the distributed algorithms and not communication issues. Therefore, this experiment has used one machine to host all PEXAs.

This initial implementation of the algorithms was designed as a simulation of process execution and recovery activities and, as such, cannot provided any definitive statements about performance measures at this stage of the research. True performance measures are affected by many factors. For example, the implementation of decentralized algorithms developed in this thesis is limited by use of existing procedures to detect write/read dependencies from (Xiao, 2006). These procedures have not been optimized for efficient retrieval of data from the local delta object schedule. This implementation is also not fully integrated into an actual process execution environment with full support for compensation or use of actual Delta-Enabled Grid Services. The main focus of this initial implementation of the algorithms was on observation of characteristics of the algorithms.

A delta generator was used to simulate every service call at a PEXA and also invokes services in other sites. The deltas generated are controlled by specifying attributes such as:

- number of processes (the number of concurrent processes)
- number of services in a process (number of composing services in a process)
- percentage of external operations
- failure rate (possible percentage for a failure to occur in a process)
- number of accessed data objects by a service invocation
- deltaProperty is set to 1 (one column in the test table is accessed)

By varying these attributes, decentralized algorithms under different situations are tested by different simulations. Data sets for different simulations are captured and used to examine algorithms in a certain level of process execution with a certain failure rate. Therefore, the changes or the potential limitations of the algorithms can be detected for performance considerations.

The following section presents the results of the analysis. The study has focused on analysis of the lazy approach, such as the graph construction and recovery. With respect to the eager algorithm, the main measurement is to examine the time to add a node to the graph after discovering data dependencies backwards for the local delta schedule. After building a graph in the eager algorithm, the recovery procedure is the same as that used by the lazy algorithm. Therefore, the distributed graph propagation based on a specific error is not measured for the eager approach. Given reported failure percentages for web services such as in [Amazon Simple Storage

Service, 2007], maintenance of the graph for successfully completed processes would be unnecessary overhead since the deletion of completed processes can lead to complicated situations as such re-structuring the graphs for adding new edges, merging nodes, and deleting nodes and edges.

6.2 Performance Analysis for the Decentralized Algorithms

This section analyzes the performance of data dependency analysis using the lazy algorithm. The major issue for the lazy algorithm is to 1) examine the average time to build local process dependency graphs, and 2) examine the number of graphs and the time for graph construction that propagates among the PEXAs as part of the recovery process. Since the eager algorithm is using the same recovery procedure to compensate dependent processes, only the time to discover dependencies and then add nodes to the graph is recorded for examination. This is indicated as average time for adding nodes per PEXA at the end of this chapter. As described in the previous subsection, the simulations were run using three PEXAs. One simulation assumed ten processes running at each PEXA, with each process executing five operations (i.e., service calls). A second simulation assumed 100 processes running at each PEXA, with the number of operation ranging from five to ten. A third simulation generated 500 processes running at each PEXA, with five to ten operations for each PEXA.

Each simulation was divided into four tests that varied parameters for the percentage of external vs. local operations, the failure rate (ranging from 2% to 20%), and the number of data items modified by one operation and number of columns. As a database, the tests used a db4o database containing ten objects and randomly generated data access patterns that ranged from 10% to 50% of the data objects. The tests assumed that each data access was based on the same column for each data item. Tables 1, 2, and 3 show the test parameters for each simulation. Each test, t1 through t4, was run ten times to generate the average number of graphs constructed per PEXA, the average local graph construction time, the total graph construction time across all

PEXAs, the average number of errors per PEXA, and the number of distributed graphs generated per error.

test id	processes	operations	% external	failure Rate	objects	column
t1	10	5	.20	.05	1~3/10	1
t2	10	5	.30	.20	1~3/10	1
t3	10	5	.50	.05	1~5/10	1
t4	10	5	.70	.10	1~5/10	1

Table 1: 10-process execution simulation

test id	processes	operations	% external	failure Rate	objects	column
t1	100	5	.20	.10	1~3/10	1
t2	100	10	.50	.10	1~3/10	1
t3	100	10	.30	.15	1~5/10	1
t4	100	10	.70	.05	1~5/10	1

Table 2 100-process execution simulation

test id	processes	operations	% external	failure Rate	objects	column
t1	500	10	.20	.10	1~5/10	1
t2	500	10	.50	.05	1~5/10	1
t3	500	10	.50	.10	1~5/10	1
t4	500	5	.70	.02	1~5/10	1

Table 3 500-process execution simulation

Figure 22 shows the relationship between the average number of graphs per PEXA and the average local graph construction time using the test parameters in Table 1 (10 processes per PEXA). Figures 23 and 24 present similar data for the 100 process simulation and the 500 process simulation, respectively.

In Figure 22, Test t1 has the smallest failure rate, but the largest percentage of local operations. As a result, the average number of graphs is small, but the graph construction time is larger since most of the data dependencies are found locally. In comparison, Test t4 has a larger percentage of external operations, with a slightly higher failure rate, generating a larger number of graphs per PEXA with a smaller graph construction time. Test t2 only accesses 30% external operations, but generates a larger number of graphs since it has the highest failure (20%) rate and there is more recovery activity than with a lower failure rate. Test t3's performance for an external percent of 50% is similar to that of Test t4 at 70%.

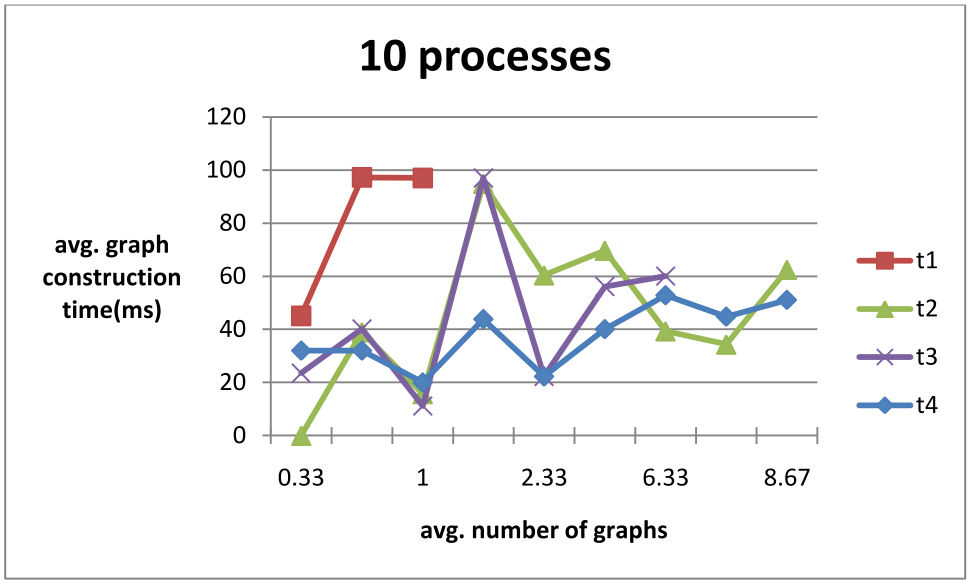


Figure 22: 10-process simulation

Figure 23 shows the results for 100 processes per PEXA. Test t1 has the lowest percentage of external operations, with more local dependencies and a higher average construction time. Test t2 has 50% internal and 50% external operations, generating more graphs with a lower construction time than t1. Test t3 is only 30% external, but has a higher failure rate (15%), generating more graphs with a smaller construction time per graph. Test t4 has the largest percentage of external operations at 70%, but it also has the lowest failure rate (5%). The number of graphs generated was smaller, with the average construction time higher than the other tests.

Figure 24 shows the results for 500 processes per PEXA. Test t1 has the lowest percentage of external operations and the highest failure rate in all, resulting in higher local dependencies and therefore a higher average construction time. Tests t2 and t3 have the same percentage of external operations (50%). The only difference is the failure rate of 5% for t2 and 10% for t3. Since t3 produced more errors and constructed slightly more graphs, the average graph construction time was lower than that of t2's.

Although Test t4 has the highest percentage of external operations, the lowest failure rate caused fewer graphs constructed and longer average graph construction time.

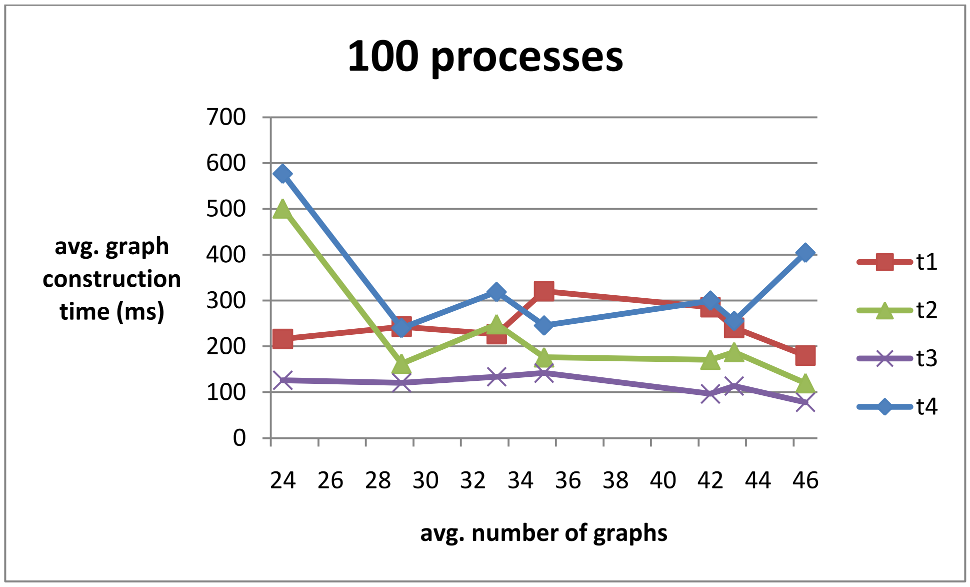


Figure 23: 100-process simulation

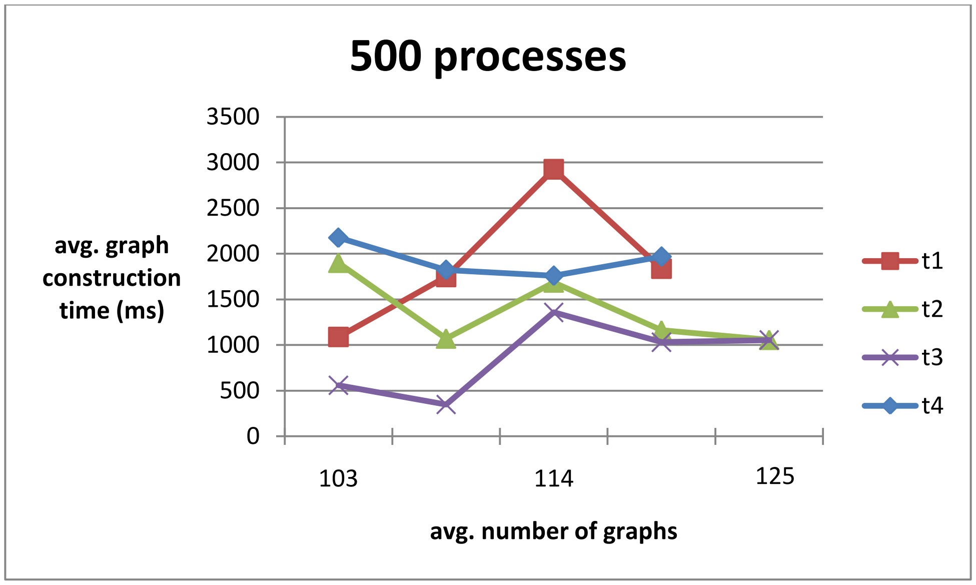


Figure 24: 500-process simulation

Therefore, with respect to Figures 22, 23, and 24, two conclusions can be drawn. The first one is that the average number of graphs is associated closely with the failure rate and percentage of external operations. A higher percentage of external operations can distribute more dependencies across PEXAs. A higher failure rate causes more errors to occur and thus more opportunities to trigger the data dependency analysis to recover dependent processes. However, a relatively lower failure rate does not necessarily mean more graphs will be generated, as in Test t4 in the 500-process simulation. The second conclusion is that the more graphs generated, the less the average graph construction time is. Since both the percentage of external vs. local operations and number of accessed data items decide the data dependencies, if the number of accessed data items is the same for two tests, the test with a lower percentage of external operations will have more local dependencies and thus have to spend more time retrieving the local delta object schedule. Therefore, more time will be consumed for the local retrieval and graph construction. More distributed dependencies are generated by the higher percentage of external operations and local dependencies might be relatively less, thus spending less time constructing the graphs but having more distributed graphs.

Figure 25 shows how many distributed graphs are generated across all three PEXAs by one specific error. Using the four tests in the three different levels of process execution, all of the average numbers from these tests are compared. The t4 tests in 100-process and 500-process simulations are the highest since they have the highest external execution rates. When a failure occurs, more graphs are generated because of more external operations from other PEXAs.

Figure 26 is the comparison between three different levels of process execution. The average numbers of errors, graphs and nodes for each level of simulation are provided for a better view. The average number of graphs is increasing significantly from 10 processes to 500 processes, while the average number of errors is increasing slightly. However, the average number of nodes per graph is not increasing significantly.

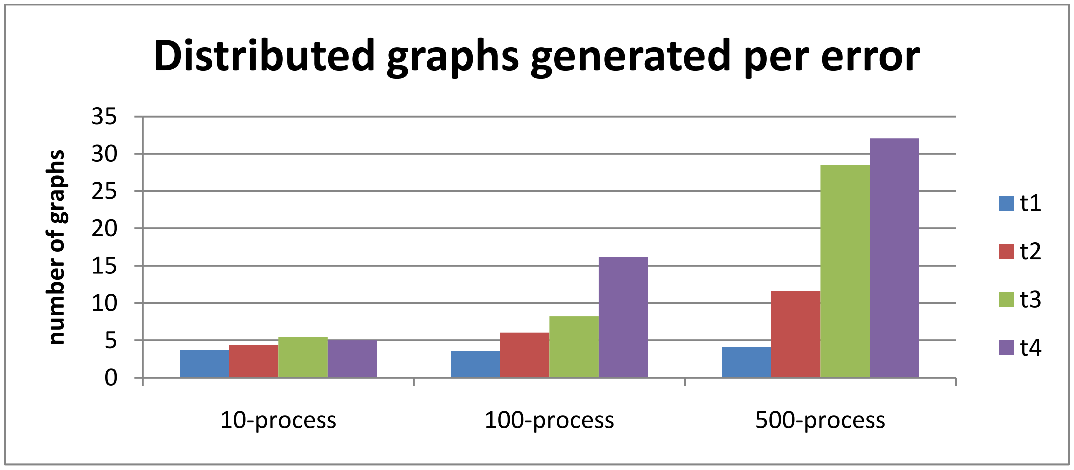


Figure 25: Distributed graphs generated per one error

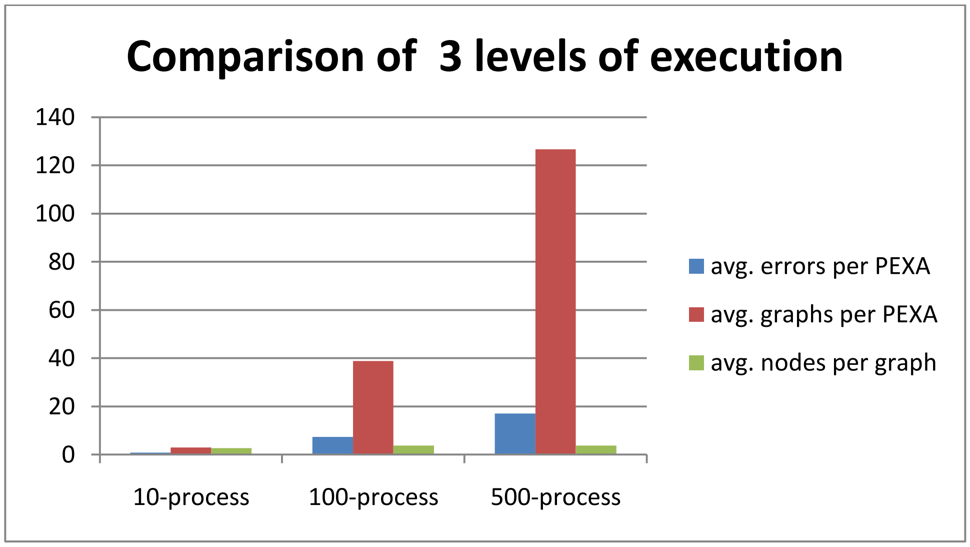


Figure 26: Comparison of 3 levels of execution

The initial implementation of eager algorithm was also tested. The Tables 1, 2, and 3 were used for the experiments. As described before, recovery process using the eager approach is the same as the lazy approach. So recovery-related information about distributed graph construction and graphs propagated per error was not recorded. Instead, the average time for adding nodes to the graph per PEXA is recorded for the examination of the performance analysis for the eager algorithm.

Figure 27 is the bar chart that shows the average time for adding nodes per PEXA using the eager algorithm. The performance of different levels of process executions is demonstrated. From 10 to 500 process execution, Tests t1, t2, and t3 are increasing slightly corresponding to different levels of execution. t4 in the 500-process level consumed relatively less time for adding nodes since the failure rate (2%) is low enough to simulate the real world applications. Therefore, Figure 27 indicates that the time to retrieve dependencies and add nodes to the graphs is relatively stable and efficient for different levels of process execution since the time of increase of similar tests is acceptable for the implementations in the industry.

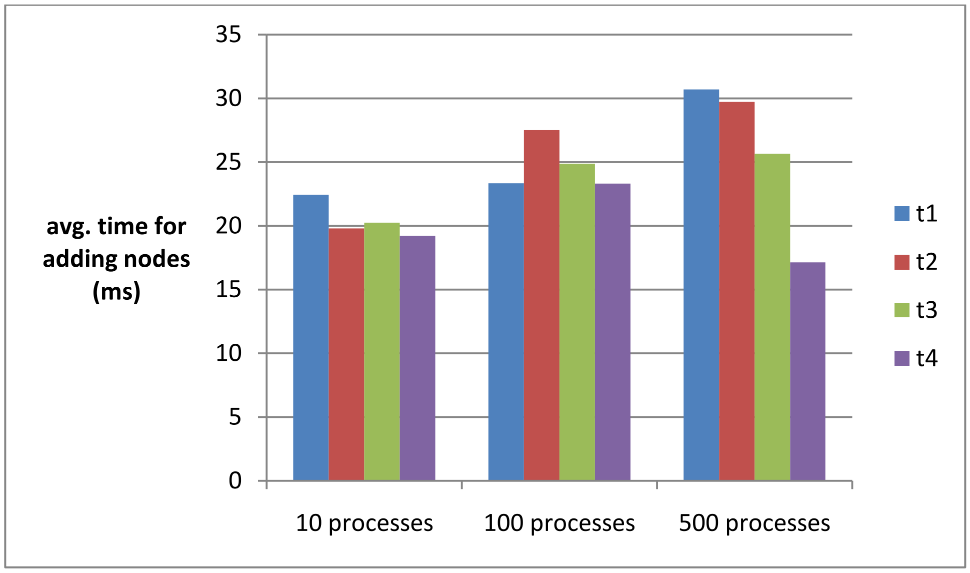


Figure 27: Average time for adding nodes per PEXA

The reason that the time for retrieving dependencies and adding nodes is less using the eager algorithm than that using the lazy algorithm is that the way to discover dependencies is different. The way to get potential read dependencies is the same for two approaches. However, the way to get write dependencies is very different. The lazy approach uses the write dependent procedure to retrieve local delta object schedule and discover dependencies often. But the eager approach only finds the latest dependencies according to modified data items provided by a set of variable in the memory. Therefore, the time to add nodes using the eager approach is less. The main overhead associated with the eager approach is maintenance of the process dependency graph for successfully completed processes. For low failure rates, the overhead may not be justified.

CHAPTER VII

SUMMARY AND FUTURE RESEARCH

This thesis has presented a decentralized approach to analyzing data dependencies among concurrently executing processes in a service-oriented environment. The decentralized approach extends existing research with the DeltaGrid project that analyzes data changes captured from service executions to identify processes that are dependent on a failed process based on data access patterns. Unlike the original work with the DeltaGrid project, where data changes are merged and analyzed in a centralized manner, this research defined algorithms that allow multiple process execution engines to share information about data dependencies. Process Execution Agents have been defined that control the execution of processes and build local delta object schedules. Process execution histories are then enhanced with control information that allows the construction of data dependency graphs to be distributed among multiple PEXAs. This research has explored a lazy algorithm that constructs distributed process dependency graphs upon the failure of a process. The research has also explored an eager algorithm that dynamically constructs process dependency graphs for all executing process so that dependency graphs are available as soon as a failure occurs. The data dependency analysis algorithms developed as part of this research represent an initial step towards the development of distributed, process-aware execution environments that can support more intelligent ways of monitoring failures, detecting dependencies, and responding to failures and exceptional conditions in an environment that cannot conform to traditional data locking protocols.

There are several directions for future research, especially considering that this work has been conducted as part of a larger project involving the development of more dynamic and flexible approaches to service composition and recovery with user-defined correctness conditions. This initial stage of the research has focused on testing and demonstrating the feasibility of the algorithms for decentralized data dependency analysis. As a result, the algorithms have not been fully integrated into an actual process execution engine. Future work should investigate the integration of the algorithms with BPEL execution engines embedded in PEXAs. The research presented in this thesis has also simplified the recovery process, assuming that all dependent processes will recover by executing compensating procedures. The use of the decentralized data dependency analysis algorithms need to be fully integrated into a service composition and recovery model, with recovery options for compensation, contingency, and retry of failed procedures (Greenfield et al., 2003; Xiao and Urban, 2009). Current research directions are defining an event and rule-based model, with user-defined correctness conditions and the ability to do partial rollbacks to checkpoints that support alternative paths for forward execution. The role of decentralized data dependency analysis in the recovery process needs to be further explored. Finally, the concept of a PEXA needs to be extended into a more process-aware execution environment that is knowledgable of the service-composition and recovery model and the manner in which it interacts with the data dependency analysis algorithm to transform PEXAs into true agents that can reason about execution and recovery among multiple PEXAs.

REFERENCES

Amazon Simple Storage Service. (2007). Website: http://aws.amazon.com/s3-sla/

Ansari, M., Ness, L., Rusinkiewicz, M., & Sheth, A. (1992). Using flexible transactions to support multi-system telecommunication applications. *Proceedings of the 18th international Conference on Very Large Data Bases* (August 23 - 27, 1992). 65-65.

Brent, R. P. (1980). An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics, 20*(2), 176-184.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., et al. (2004). Web services architecture. *W3C Working Group Note, 11*.

Cabrera, F., Copeland, G., Cox, B., Freund, T., Klein, J., Storey, T., et al. (2002a). Web services transaction (WS-transaction). *Microsoft, IBM etc*.

Cabrera, F., Copeland, G., Freund, T., Klein, J., Langworthy, D., Orchard, D., et al. (2002b). Web services coordination (WS-Coordination). *joint specification by BEA, IBM, and Microsoft, Aug*.

Eder, J., & Liebhart, W. (1995). The workflow activity model WAMO. *Proc. of the 3rd Int. Conference on Cooperative Information Systems (CoopIs)*.

Floyd, R. W. (1967). Nondeterministic Algorithms. *J. ACM, 14*(4), 636-644.

Foster, I., Kishimoto, H., Savva, A., Berry, D., Djaoui, A., Grimshaw, A., et al. (2004). The open grid services architecture. *The Grid2: Blueprint for a New Computing Infrastructure*, 215-257.

Garcia-Molina, H., & Salem, K. (1987). Sagas. *Proc. of the ACM SIGMOD Annual Conference on Management of Data*, 249-259.

Greenfield, P., Fekete, A., Jang, J., & Kuo, D. (2003). Compensation is not enough. *7th Int. Conf. on Enterprise Distributed Object Computing.*

Greenfield, P., Fekete, A., Jang, J., Kuo, D., & Nepal, S. (2007) Isolation support for service-based applications: A position paper. *Proc. of CIDR*.

Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., et al. (2007). Web services business process execution language version 2.0. *OASIS Standard, 11*.

Kamath, M., & Ramamritham, K. (1998). Failure handling and coordinated execution of concurrent workflows. *Proc. of the IEEE Int. Conference on Data Engineering*, 334-341.

Mikalsen, T., Tai, S., & Rouvellou, I. (2002). Transactional attitudes: Reliable composition of autonomous Web services. *Workshop on Dependable Middleware-based Systems (WDMS 2002).*

Newcomer, E.; Robinson, I.; Freund, T.; Green, A.; Harby; Little, M. (2006). Web Services Business Activity (WS-Business Activity).

Paterson, J., Edlich, S., Hörning, H., & Hörning, R. (2006). *The Definitive Guide to db4o*: Apress Berkely, CA, USA.

Singh, M. P., & Huhns, M. N. (2005). *Service-oriented computing: semantics, processes, agents*: Wiley.

Tumma, M. (2004). *Oracle Streams: High Speed Replication and Data Sharing*: Rampant TechPress.

Urban, S. D., Xiao, Y., Blake, L., & Dietrich, S. W. (2009a). Monitoring Data Dependencies In Concurrent Process Execution through Delta-Enabled Grid Services. *International Journal of Web and Grid Services, 5*(1), 85-106.

Urban, S. D., Liu, Z., & Gao, L. (2009b). Decentralized Data Dependency Analysis for Concurrent Process Execution. *Middleware for Web Service Workshop, Auckland*, New Zealand.

Weikum, G. (1991). Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems (TODS), 16*(1), 132-180.

Worah, D., & Sheth, A. (1997). Transactions in transactional workflows. *Advanced Transaction Models and Architectures*, 3-34.

Wächter, H., & Reuter, A. (1992). *The contract model*: Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

Xiao, Y. (2006). Using deltas to analyze data dependencies and semantic correctness in the recovery of concurrent process execution. *Ph.D. Dissertation*, Arizona State Univ., Tempe, AZ

Xiao, Y., & Urban, S. D. (2008a). *Using Data Dependencies to Support the Recovery of Concurrent Processes in a Service Composition Environment (CoopIs)* , *Monterrey, Mexico*, 139-156.

Xiao, Y., & Urban, S. D. (2008b). Process Dependencies and Process Interference Rules for Analyzing the Impact of Failure in a Service Composition Environment. *Journal of Information Science and Technology,* 5(2), 21-45.

Xiao, Y., & Urban, S. D. (2009). The DeltaGrid Service Composition and Recovery Model. *International Journal of Web Services Research*, 6(3), 35-66.

Zhao, W., Moser, L. E., & Melliar-Smith, P. M. (2005). A reservation-based coordination protocol for Web Services. *Proceedings of the IEEE International Conference on Web Services*, 49-56.