Decentralized Communication for Data Dependency Analysis Among Process Execution Agents

Susan D. Urban, Ziao Liu, Le Gao
Texas Tech University
Department of Computer Science
Lubbock, TX 79409
susan.urban@ttu.edu, ziao.liu@ttu.edu, le.gao@ttu.edu

ABSTRACT:

This paper presents our results with the investigation of decentralized data dependency analysis among concurrently executing processes in a service-oriented environment. Distributed Process Execution Agents (PEXAs) are responsible for controlling the execution of processes that are composed of web services. PEXAs are also associated with specific distributed sites for the purpose of capturing data changes that occur at those sites in the context of service executions using Delta-Enabled Grid Services. PEXAs then exchange this information with other PEXAs to dynamically discover data dependencies that can be used to enhance recovery activities for concurrent processes that execute with relaxed isolation properties. This paper outlines the functionality of PEXAs, describing the data structures, algorithms, and communication mechanisms that are used to support decentralized construction of distributed process dependency graphs, demonstrating a more dynamic and intelligent approach to identifying how the failure of one process can potentially affect other concurrently executing processes.

KEY WORDS:
Service composition, concurrent processes, data dependency analysis, process execution agents, decentralized communication, failure and recovery

INTRODUCTION

One of the advantages of service-oriented computing is that it allows business processes to be composed by executing distributed web services (Jordan, Evdemon et al., 2007). Unlike traditional distributed transaction processing, however, since each service 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, processes composed of web services do not generally execute as transactions that conform to the concept of serializability. Since a service can commit before a global process is complete, dirty reads and dirty writes can occur among globally executing processes.

From an application point of view, dirty reads and dirty writes do not necessarily indicate an incorrect execution, and a relaxed form of correctness dependent on application semantics can produce better throughput and performance. User-defined correctness of a process can be specified as in related work with advanced transaction models (Rolf, Klas et al., 1998) and transactional workflows (Worah and Sheth, 1997), using concepts such as compensation to semantically undo a process. But even when one process determines that it needs to execute compensating procedures, information about global data dependencies is needed to determine how the data changes caused by the recovery of one process can possibly affect other processes that have either read or written data modified by the services of the failed process. This ability to capture and analyze data dependencies in a service composition environment does not exist in

current service-oriented architectures, thus creating data consistency problems for concurrent execution and limiting the effectiveness of recovery procedures for failed processes.

This paper presents our results with the investigation of an approach that performs decentralized data dependency analysis among concurrently executing processes in a service-oriented environment. In particular, we present 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.

The ability to capture data changes, known as *deltas*, builds on our past work with the use of Delta-Enabled Grid Services (DEGS) (Blake 2006; Urban, Xiao et al., 2009a). DEGS are Grid Services that have been extended with the capability of recording and externalizing incremental data changes using features such as Oracle Streams (Tumma, 2004). Whereas the work in (Urban, Xiao et al., 2009a; Xiao, 2006; Xiao and Urban, 2008a) forwarded streaming deltas from multiple DEGS to a single, time-ordered, delta object schedule for a centralized approach to data dependency analysis, the work presented in this paper has extended the data dependency analysis process to support decentralized communication among multiple PEXAs. Each PEXA creates its own local delta object schedule that can be used to create process dependency graphs. But since a process can execute services that are associated with multiple PEXAs, the data dependency analysis process requires a global view of distributed process dependency graphs.

This paper outlines the functionality of PEXAs and also describes the data structures, algorithms, 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 (Liu, 2009; Urban, Liu et al., 2009b). 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. One of the challenges with building distributed process dependency graphs is discovering the *hidden dependencies* that are created by having each PEXA maintain its own local delta object schedule. We describe the use of *link objects* and other runtime control information that is used to discover hidden dependencies as well as global cycles in the construction of distributed process dependency graphs. We also illustrate how the distributed graphs are used to propagate recovery procedures, with a simulation and evaluation that addresses graph construction time and the number of sub-graphs generated within a distributed dependency graph.

The decentralized approach eliminates the bottleneck and overhead reported in (Urban, Xiao et al., 2009a) of forwarding all data changes to a central point for analysis. More importantly, the distributed delta object schedule and decentralized data dependency algorithms described in this paper 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 paper, we first present related work. We then outline the functionality of PEXAs together with an illustration of graph construction issues for decentralized data

dependency analysis. We then present the algorithms for the lazy and eager approaches to decentralized data dependency analysis, followed by an evaluation of the algorithms in a simulation environment. The paper concludes in with a summary and discussion of future research directions.

Related Work

This section first summarizes past work with advanced transaction models and transactional workflows. We then address recent work with recovery issues for service-oriented. The section concludes by presenting background on our own past research with the DeltaGrid project that provides the basis for the work described in this paper.

Advanced Transaction Models

Advanced Transaction Models (ATMs) were designed to relax traditional ACID properties and the use of the two-phase commit protocol to provide functionalities such as compensation for backward recovery and contingency for forward recovery. ATMs providebetter support for Long Running Transactions (LRTs) that need relaxed atomicity and isolation properties (Cichocki, Helal et al., 1998). In the work of (Garcia-Molina and Salem, 1987), sagas were defined as a mechanism to structure long running processes. A saga defines a chain of transactions, with each sub-transaction having a compensating procedure to reverse the affects of the saga when it fails. Other advanced transaction models, such as the multi-level transaction model and the flexible transaction model have made use of compensation for hierarchically structured transactions (Rolf, Klas et al., 1998). In fact, current standards for web services, such as WS-BPEL (Jordan, Evdemon et al., 2007) and WS-Business Activity (Newcomer, Robinson et al., 2006) build on the concept of compensating procedures as a means of recovery. These models, however, do not support isolation of data and do not address recovery for dependent transactions in loosely-coupled applications.

Transactional Workflows

The term transactional workflows was introduced to recognize the relevance of transactions to workflow activity. Unlike database transactions, transactional workflows do not support all ACID properties. Transactional workflows involve the coordinated execution of multiple related tasksthat 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 provides a classic example of work with transactional workflows (Wächter and Reuter, 1992), supporting the correct execution of non-atomic, long-lived applications with application-dependent consistency constraints. The ConTract Model provides compensation for backward recovery, and user-defined consistency through the specification of pre-conditions or post-conditions for steps. Other examples of transactional workflow models include the Workflow Activity Model (Eder and Liebhart, 1995), the Crew Project (Kamath and Ramamritham, 1998), and METEOR (Worah and Sheth, 1997).Transactional workflow models have improved the robustness of distributed transaction executions, but the work in this area still does not address the affect that a failed process can have on other concurrently executing processes. Most of the work in this area depends on declared knowledge of workflow dependencies.

Workflow management systems have been studied in the context of transaction workflows. Workflow Management Systems typically provide exception handlers to support backward and

forward recoveries (Chiu, Li et al., 2000; Hagen and Alonso, 2000; Kiepuszewski, Muhlberger et al., 1998). However, correctness in workflow management is still an open question, especially with respect to data consistency issues (Kamath and Ramamritham, 1996).

Transactional Issues for Service Compositions

Numerous other techniques are being investigated for addressing data consistency in service composition. Tentative holding is used in (Limthanmaphon and Zhang, 2004) to achieve a tentative commit state for transactions over Web Services. Acceptable Termination States (ATS) (Bhiri, Perrin et al., 2005) are used to ensure user-defined failure atomicity of composite services. The concept of a promise is used in (Greenfield, Fekete et al., 2007), where a promise is an agreement between a client and a resource owner, allowing a service provider to offer assurances that resources will be available when they are needed. A reservation-based protocol is defined in (Zhao, Moser et al., 2005) where a process uses an explicit reservation phase to request resources, followed by an explicit confirmation/cancellation phase. By borrowing the trigger feature in active database, compensation rules are defined for all subtransaction operations in (Strandenæs and Karlsen, 2002). To enhance compensation flexibility, work such as that in (Lin and Liu, 2005) has used business rules to alter the execution of compensating procedures. Other techniques include Web Services Composition Action (Tartanoglu, Issarny et al., 2003), WebTransact (Pires, Benevides et al., 2003), and the work of (Vidyasankar and Vossen, 2004), defining a model that supports features such as atomic transactions, pivot transactions, compensatable transactions, and re-triable transactions, as well as forward and backward recovery techniques.

In the above techniques, the question of how the recovery of a composite process could possibly affect other concurrently executing processes has not been addressed. The technique presented in this paper 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.

The DeltaGrid Project

The research described in this paper builds on our past work with the DeltaGrid project (Xiao, 2006; Xiao and Urban, 2008a; Xiao and Urban, 2008b) and Delta-Enabled Grid Services (DEGS) (Blake 2006; Urban, Xiao et al., 2009a). A DEGS is a Grid Service that has been enhanced with an interface that provides access to the incremental data changes, or deltas, that are associated with service execution in the context of globally executing processes. A DEGS uses an OGSA-DAI Grid Data Service for database interaction. The database captures deltas using capabilities provided by most commercial database systems. In (Urban, Xiao et al., 2009a), we experimented with triggers and with the use of Oracle Streams as a way to capture data changes. 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 (Tuma, 2004).

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, where 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 schedule of data changes from distributed DEGS. This global delta object schedule is

used to support recovery activities when process execution fails (Xiao and Urban, 2008a). The global delta object schedule can be used to support the backward recovery of a completed service and also provides the basis for discovering data dependencies among processes. 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. The work in (Xiao and Urban, 2008b) describes a technique that uses user-defined rule conditions to determine the recovery actions of processes that are dependent on a failed process.

Our past research results have 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 major bottleneck in the process. The results presented in this paper extend the data dependency analysis concept to a decentralized approach, 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.

Process Execution Agents (PEXAs)

This section provides an initial overview of process execution agents. The discussion begins with an example execution scenario in Figure 1, where we assume 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 lightening 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. This section presents an example execution scenario, describes the internal architecture of a PEXA, and outlines challenges for decentralized data dependency analysis.

A PEXA Execution Scenario

As shown in Figure 1, 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_3controls 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 1, 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.

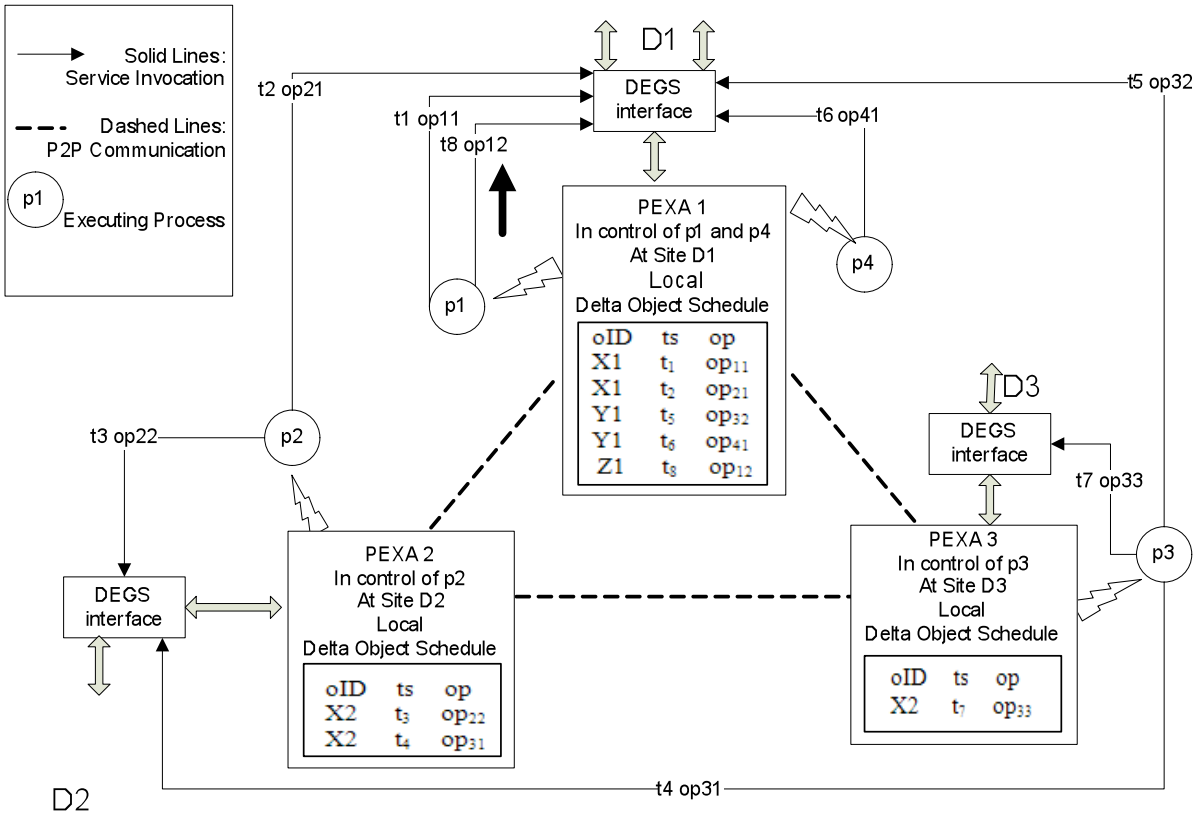

Figure 1: Decentralized Process Execution Agents

Internal PEXA Architecture

Figure 2 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, Edlich 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 an indexing structure defined in (Xiao, 2006) 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 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. The next subsection elaborates on the use of *link objects* and other runtime information to construct global, distributed process dependency graphs.

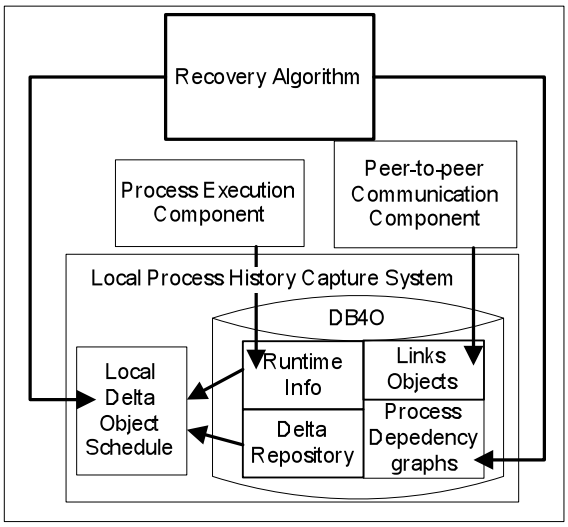

Figure 2: Internal PEXA Architecture

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 paper 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 originally defined in (Xiao, 2006), 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.

As an example, consider again the process execution scenario in Figure 1. Figure 3 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 part of Figure 4, 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 3 and 4together 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 4, 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 4 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 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 1, 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 4. 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}).

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 4, 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. The following section elaborates on the algorithm for constructing distributed process dependency graphs among decentralized PEXAs.

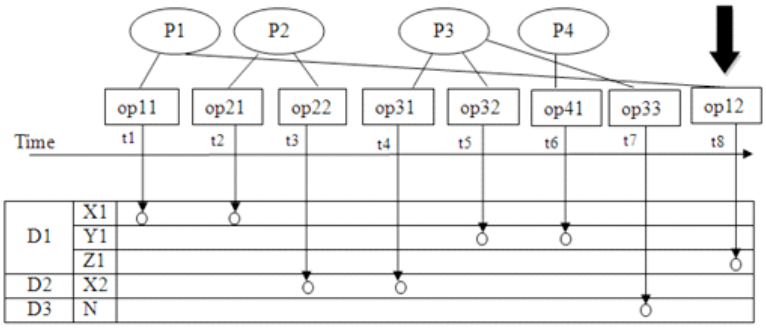

Figure 3: Data Access View of Interleaved Execution

Dependency Analysis Using the Lazy Algorithm

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. This sectionaddresses the lazy algorithm (Urban, Liu et al., 2009b), describing dependency graph construction under the lazy approach and propagation of the recovery process among multiple PEXAs. The eager algorithm is addressed in the following section, describing dependency graph construction during process execution. Both algorithms are demonstrated using the execution scenario from Figure 1.

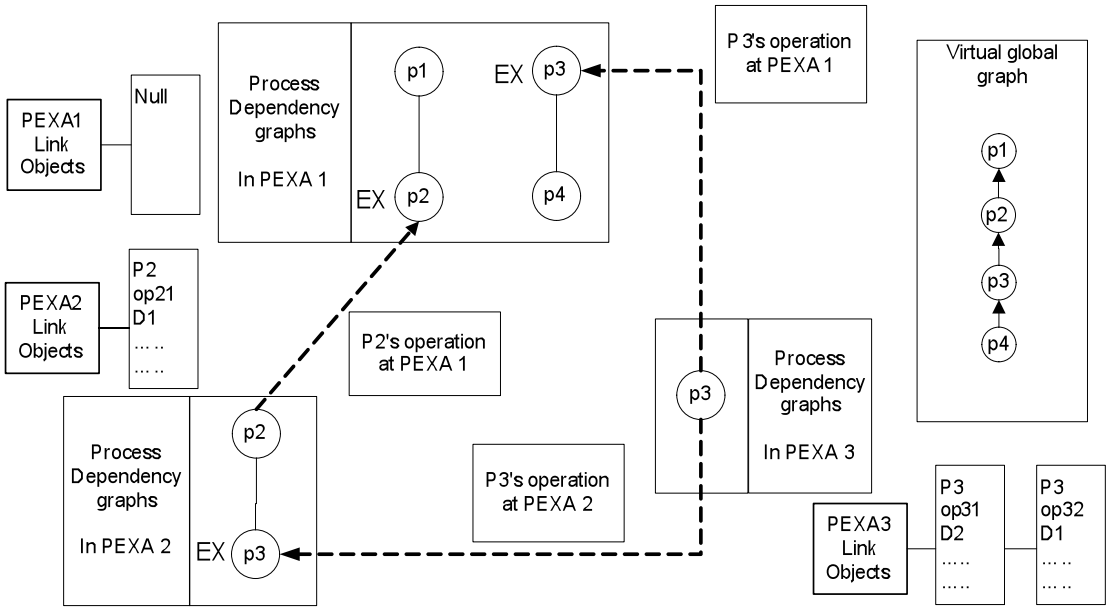

Figure 4: Global, Distributed Process Dependency Graph

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.

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.

There can potentially be cycles in a process dependency graph. For example, 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 before p_3 is recovered. Here, compensation is used as a logical approach to backward recovery. 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) 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) 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 that defines the current value of the dependency graph using a breadth first traversal. The traversal() method is used to do a breadth first traversal of the graph and return the value of result.

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 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.

The Lazy Algorithm

Figure 5 provides pseudocode of the graph propagation for the lazy algorithm. This procedure is called after the PEXA discovers a failed process 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.

```
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 5: findProcessDependencies() Procedure

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.

```
// 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(inti=0;i<pList.size();i++)
{
//use a temporal variable
ProcessInfotemP=(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 6: buildGraph() Procedure

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 6, 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 7 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.

```
// 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 7: recover() 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.,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.

Execution Scenario for the Lazy Algorithm

Figure 8 uses the execution scenario from Figure 1 to illustrate the logic of the algorithm presented in Figures 5-7. 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 8. Notifications that are initiated by the sendNotification() procedure are drawn as dashed lines in Figure 8.

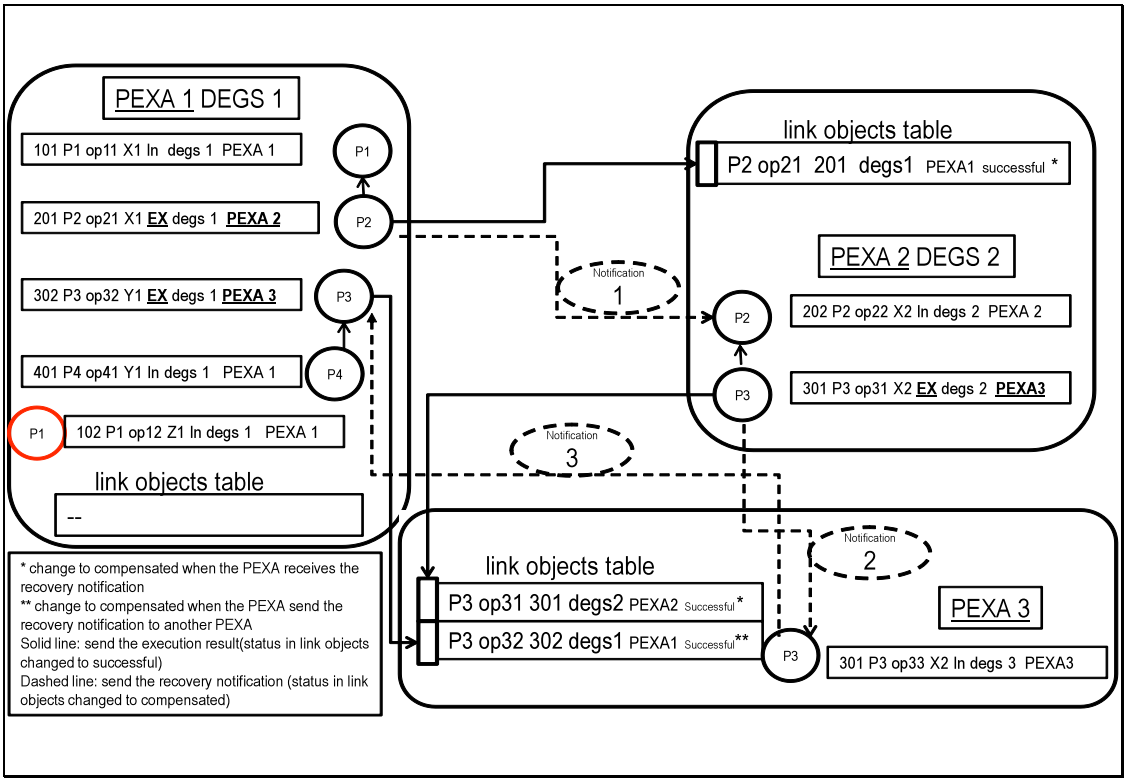

Figure 8: Lazy Algorithm Execution Scenario

In the scenario from Figure 4, 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 8, 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 8), 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)).

At PEXA 2, the graph $p_2 \leftarrow p_3$ is created from the local delta object schedule. The algorithm in Figure 7 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 8, successful* is changed to compensated for op_{21} in thePEXA 2link 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 8) 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 7 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 8). 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.

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.

The Eager Algorithm

Figure 9 illustrates the difference between the lazy and eager algorithms. As mentioned in the previous section, 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.

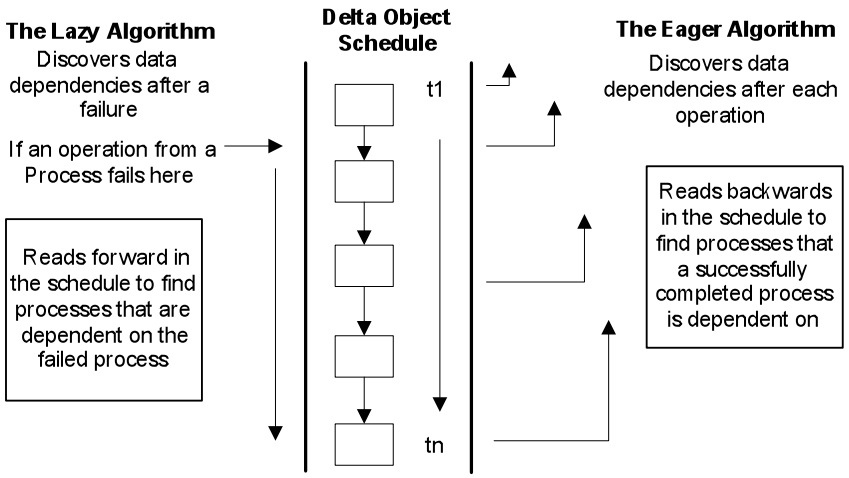

Figure 9: Difference between Lazy and Eager Algorithm

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, especially if the process failure rate is low.

Figure 10 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 11 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
booleannodeInGraph=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(inti=0;i<merge.size();i++)
g.addEdge((String)merge.get(i), processId);
         }
}
```

Figure 10: 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(inti=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 11: checkLastModificationOnSameDataItem()Procedure

The eager approach assumes that a process will potentially fail and collects all the dependencies to build 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 7 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.

Execution Scenario for the Eager Algorithm

The example in Figure 12shows 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 p1, p2, and p3. The right side of the figure shows the dependency graph that is constructed along with the process execution at runtime.

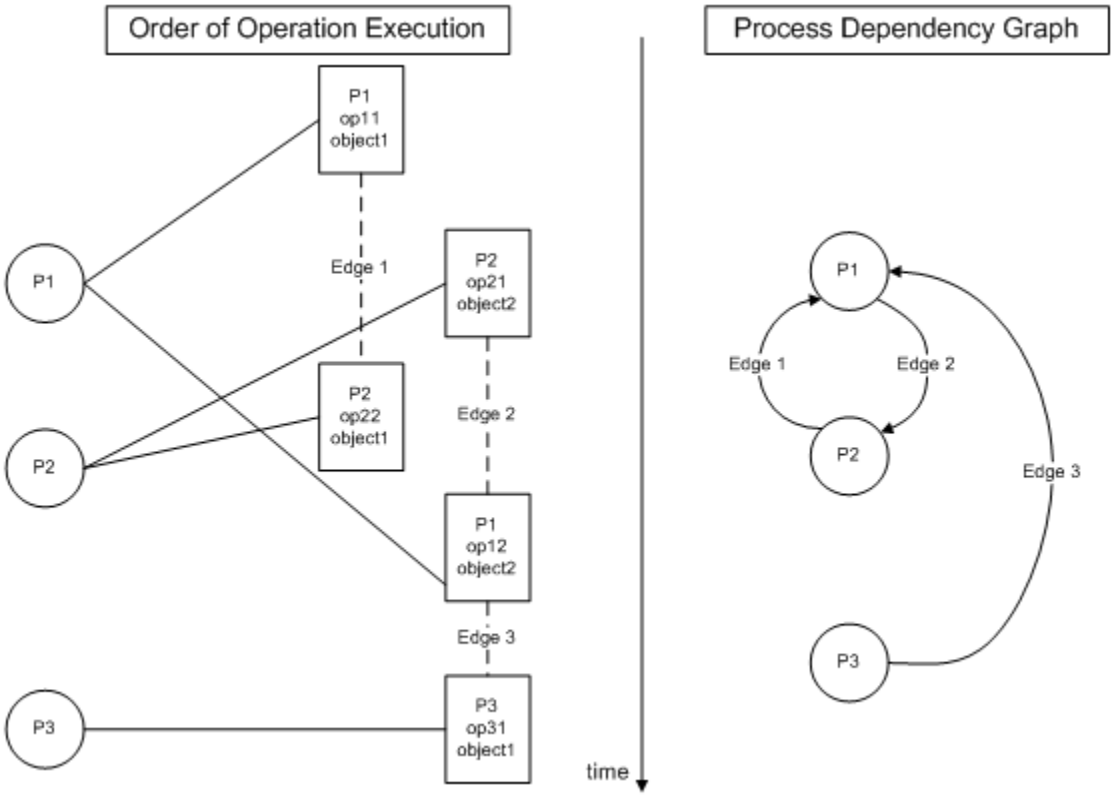

Figure 12: Process Dependency Graph Construction with the Eager Approach

In Figure 12, 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 executingop_{21} of p_2, based on the data items 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 12.

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 12, is also added, creating a cycle in the graph. Unlike the lazy algorithm, cycles are needed in the graph since the order of dependent processes to be compensated depends on which process fails.

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 12.

Decentralized Scenario for the Eager Algorithm

Figure 13uses the distributed execution scenario from Figure 4, 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 latestOperationOnDatais 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 latestOperationOnDatavariable, 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.

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.

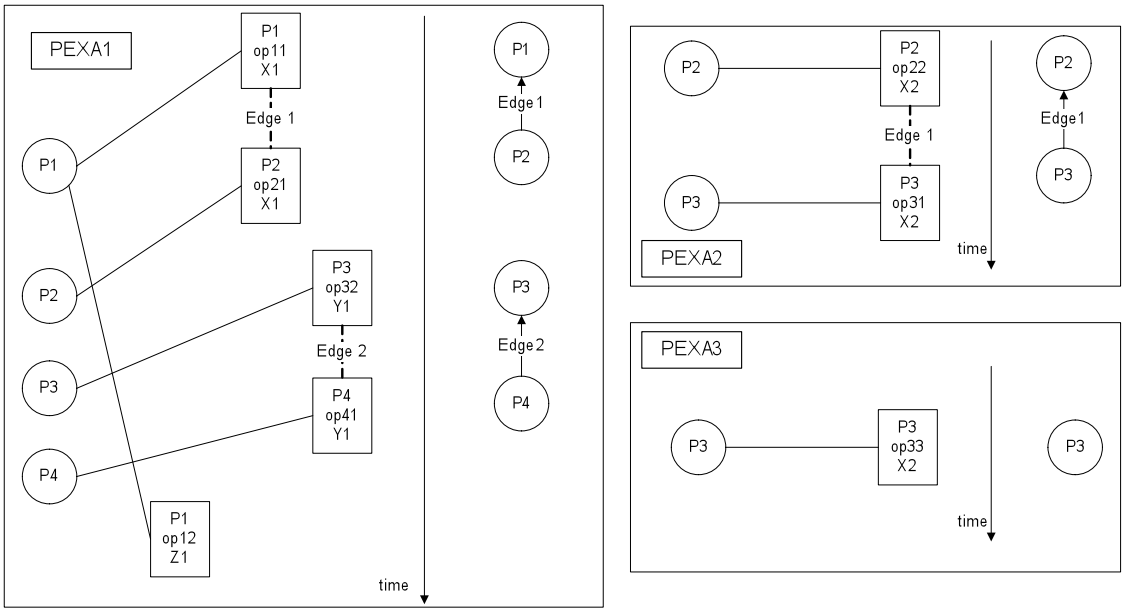

Figure 13: Decentralized Execution Scenario for the Eager Algorithm

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 7 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.

Process Dependency Graph Issues for the Eager Algorithm

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 12 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. Therefore, when a failure occurs, the graph is traversed to create the recovery order of dependent processes.

The primary overhead issue for the eager algorithm is maintenance of the graph during execution. It is important to delete nodes from the process dependency graph to prevent the graph from growing unnecessarily large with continuously executing processes. There are two situations to consider for deletion of nodes. One situation occurs after the execution of the recovery procedure when sub-graphs need to be deleted from the graph structure for recovered processes. The other situation requires that PEXAs using the eager algorithm periodically examine their own graphs

for the deletion of completed processes. The work in (Liu, 2009) elaborates on both of these situations for the eager algorithm.

Implementation and Evaluation of Decentralized Data Dependency Analysis

This section presents an evaluation of the decentralized data dependency analysis algorithms. The first subsection describes the implementation environment and measurement criteria. The second subsection presents an evaluation of the recovery propagation algorithm that is central to the lazy and eager algorithms.

Implementation Environment and Measurement Criteria

The implementation was done on a workstation using 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. This research mainly focused on analyzing different aspects of the distributed algorithms rather than communication costs. As a result, all PEXAs were deployed on the same workstation.

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 were originally designed to demonstrate the functionality of the delta object schedule and have not been optimized for efficient retrieval of data. 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

By varying these attributes, decentralized algorithms under different situations were tested under different simulations. Data sets for different simulations were captured and used to examine the algorithms with different process failure rates. The study has focused on analysis of the lazy approach with respect tograph construction and recovery. For the eager algorithm, the main measurement is to examine the time to add a node to the graph after discovering data dependencies 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.

Performance Analysis for the Decentralized Algorithms

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. As described in the previous subsection, the simulations were run using three PEXAs. One simulation assumed 10 processes running at each PEXA, with each process executing five operations (i.e., service invocations). A second simulation assumed 100 processes running at each PEXA, with the number of operations ranging from five to ten. A third simulation generated 500 processes running at each PEXA, with five to ten operations for each PEXA.

Figure 14 shows the number of graphs vs. graph construction time for 100 processes per PEXA. Results for the 10 and 500 process simulations can be found in (Liu, 2009). Each simulation was divided into four tests:

- t1 represents 20% external operations, 5-10% failure rate, and random access to 30-50% of the data objects,
- t2represents 30-50% external operations, 5-20% failure rate, with random access to 30-50% of the data objects,
- t3 represents 30-50% external operations, 5-15% failure rate, with random access to 50% of the data objects, and
- t4 represents 70% external operations, a 2-10% failure rate, with random access to 50% of the data objects.

The percentage of external operations increases from 20% in t1 to 70% in t4. The percentage of data objects accessed also increases from a range of 30-50% in tests t1 and t2 to 50% in tests t3 and t4. Test t2 also has the largest failure rate, while t4 has to lowest failure rate.

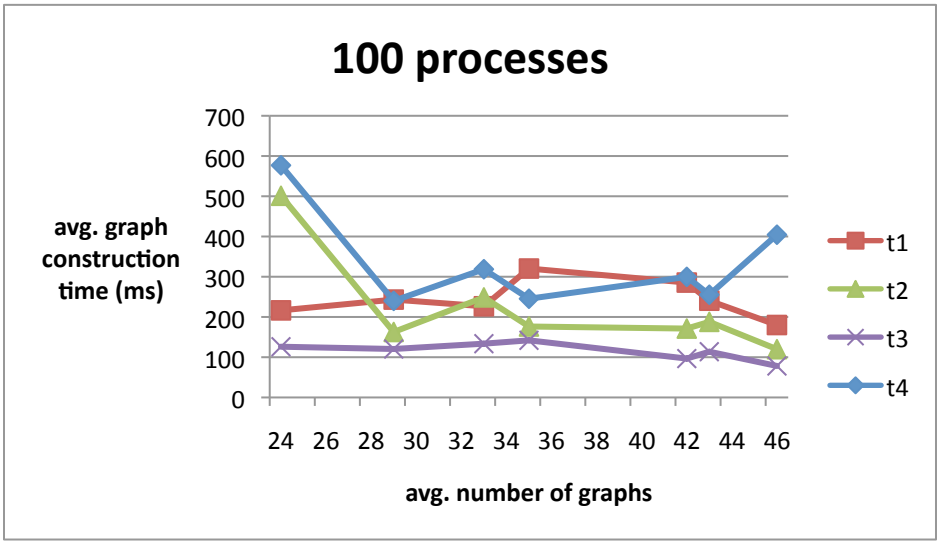

Figure 14: 100-process simulation

One common observation from the 10, 100, and 500 processsimulation is that the average number of graphs generated by the algorithm is closely associated 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 also causes more errors to occur and thus more opportunities to trigger the data dependency analysis to recover dependent processes. A second observation is

that the more graphs generated, the less the average graph construction time. 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 of individual graphs. More distributed dependencies are generated by the higher percentage of external operations. As a result, local dependencies might be less, thus spending less time constructing each local graph, but having more distributed sub-graphs to build.

Figure 15 shows the number of distributed graphs generated across all three PEXAs by a single error for the 10-process, 100-process, and 500 process simulations. The t4 tests are the highest in the 100 and 500 process simulations since they have the highest external operation rates. Figure 16 compares the average numbers of errors, graphs and nodes for each simulation. The average number of graphs increases significantly from 10 processes to 500 processes, while the average number of errors increases slightly. However, the average number of nodes per graph does not increase significantly as the number of processes per PEXA increases.

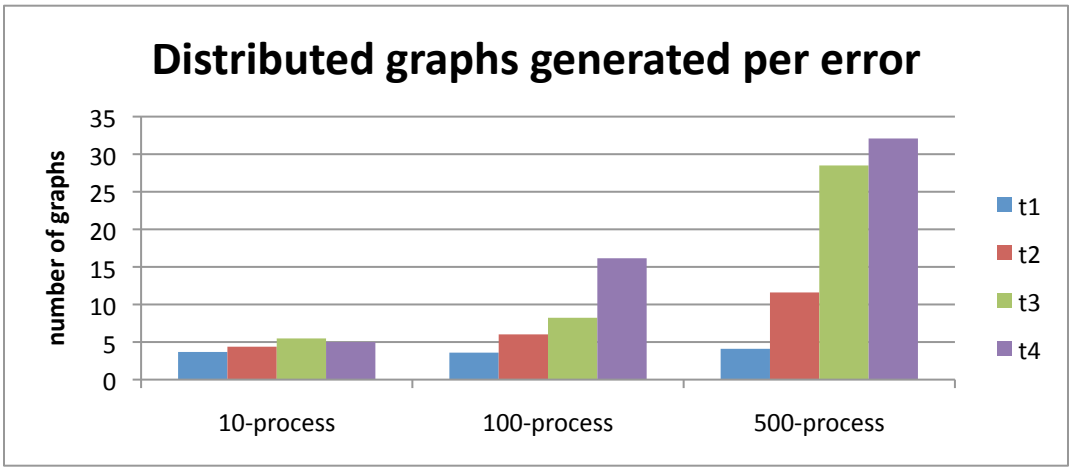

Figure 15: Distributed graphs generated per one error

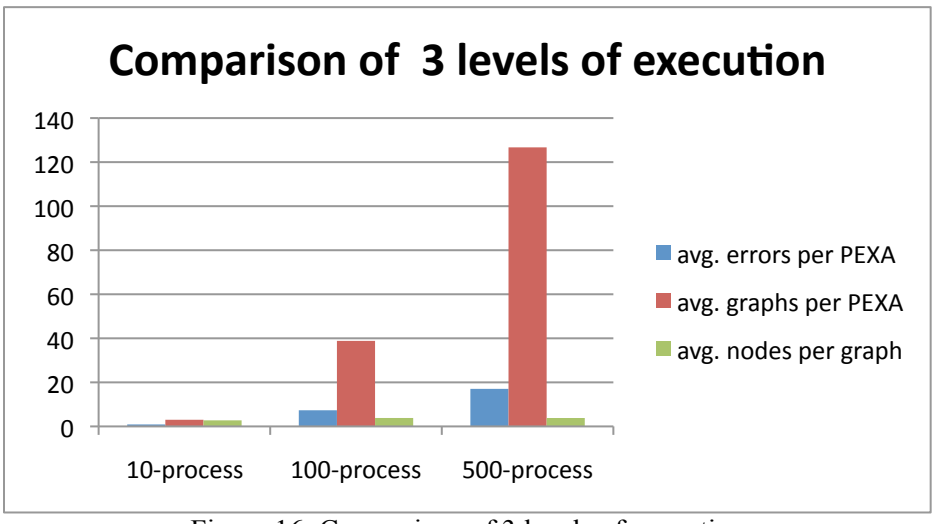

Figure 16: Comparison of 3 levels of execution

Figure 17 shows the average time for adding nodes per PEXA using the eager algorithm. For the 10, 100, and 500 process execution, tests t1, t2, and t3 increase slightly corresponding to the

different levels of execution. Test t4 in the 500-process simulation consumed less time for adding nodes since the failure rate (2%) was low. The time to retrieve dependencies and add nodes to the graphs is relatively stable for the eager algorithm and, in fact, is less than the time for retrieving dependencies using the lazy algorithm since the process for discovering dependencies is different for each technique. The lazy approach searches through a larger collection of deltas to discover dependencies. The eager approach, however, only finds the latest dependencies according to modified data items provided by a set of variables in 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. Given the low percentages for web service failures as reported 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 re-structuring the dependency graph for adding new edges, merging nodes, and deleting nodes and edges.

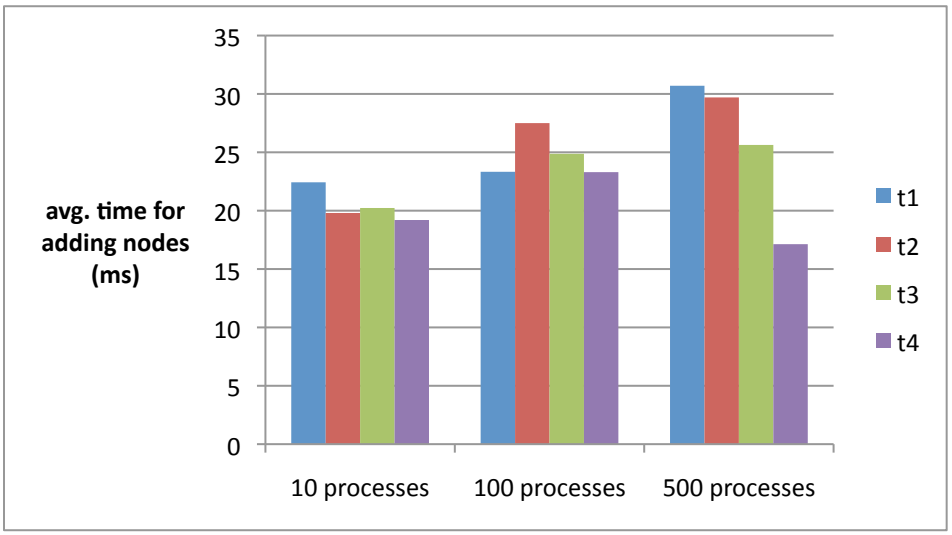

Figure 17: Average time for adding nodes per PEXA

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, Fekete 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.

ACKNOWLEGMENT

This research has been supported by the National Science Foundation under Grant No. CCF-0820152. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

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

Bhiri, S., Perrin, O., & Godart, C. (2005).Ensuring required failure atomicity of composite web services.*Proc. of the 14th Int. Conference on World Wide Web*, 138-147.

Blake, L. (2006). The Design and Implementation of Delta-enabled Grid Services, M.S. Thesis, Arizona State University, 2006.

Cichocki, A., Helal, A.,Rusinkiewicz, M., & Woelk, D. (1998).*Workflow and process automation concepts and technology*: Kluwer Academic Publishers.

Chiu, D., Li, Q., & Karlapalem, K. (2000). Facilitating exception handling with recovery techniques in ADOME workflow management system.*Journal of Applied Systems Studies,* 1, 467-488.

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

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*.

Hagen, C., & Alonso,G. (2000). Exception handling in workflow management systems.*IEEE Transactions on Software Engineering*, 26, 943-958.

Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., & Goland, Y. (2007). Web services business process execution language version 2.0. OASIS Standard, 11.

Kamath, M., & Ramamritham, K.(1996). Correctness issues in workflow management. *Distributed Systems Engineering,* 3, 213-221.

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

Kiepuszewski, B., Muhlberger, R., & Orlowska, M. (1998). FlowBack: providing backward recovery for workflow management systems.*ACM SIGMOD Record,* 27, 555-557.

Limthanmaphon, B., & Zhang, Y. (2004). Web service composition transaction management.*Proc. of the 15th Australasian Database Conference*, 171-179.

Lin, L., & Liu, F. (2005). Compensation with dependency in web services composition.*Proceedings of the International Conference on Next Generation Web Services Practices*, 183-188.

Liu, Z. (2009). Decentralized Data Dependency Analysis for Concurrent Process Execution, M.S. Thesis, Department of Computer Science, Texas Tech University, Fall 2009.

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*, Berkely, CA, USA.

Pires, P. F., Benevides, M. R. F., & Mattoso, M. (2003). Building reliable web services compositions.*Lecture Notes in Computer Science: Web, Web Services, and Database Systems*, 59-72.

Rolf, A., Klas, W., & Veijalainen, J.(1998).*Transaction management support for cooperative applications*: Kluwer Academic Publishers.

Strandenæs, T., & Karlsen, R. (2002).Transaction compensation in web services.*NorskInformatikkonferanse*.

Tartanoglu, F., Issarny, V.,& Romanovsky, A. (2003).Coordinated forward error recovery for composite web services. *Proc. of the IEEE Symposium on Reliable Distributed Systems*, 167-176.

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.

Vidyasankar, K., & Vossen, G. (2004). A multilevel model for Web service composition.*Proc. of the IEEE Int. Conference on Web Services*, 462-469.

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, USA.

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.

ABOUT THE AUTHOR(S)

Susan D. Urban received the B.S, M.S., and Ph.D. degrees in computer science in 1976, 1980, and 1987, respectively, from the University of Louisiana at Lafayette. She is currently a Professor in the Department of Computer Science at Texas Tech University. She was previously at Arizona State University from 1989-2007, where she currently holds the status of Emeritus Professor. She was also an Assistant Professor at the University of Miami from 1987-1989.Her research addresses event processing as well as integrated techniques for event, rule, and transaction processing to address data consistency and active behavior in distributed, data-centric applications.

Ziao Liu received the B.E. degree in computer science in 2007 from Taishan University in China and the M.S. degree in 2009 from Texas Tech University at Texas.

Le Gao received his B.S. degree in computer science in 2004 from Nanjing University of Aeronautics and Astronautics, China. He is currently a Ph.D. student in the department of computer science at Texas Tech University. Before coming to Texas Tech University, he was a database engineer at China Mobile corporation from 2006-2007. He was also working at China Telecom corporation as a database operator from 2004-2006.