Decentralized Data Dependency Analysis for Concurrent Process Execution

Susan D. Urban, Ziao Liu, Le Gao
Texas Tech University
Edward E. Whitaker Jr. College of Engineering
Department of Computer Science
Lubbock, TX 79409
susan.urban@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 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.**

Keywords-data dependency; concurrent process execution; relaxed isolation; process recovery; decentralized communication

I. INTRODUCTION

One of the advantages of service-oriented computing is that it allows business processes to be composed by executing distributed web services [16]. 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 [12] and transactional workflows [21], 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, also known as *deltas*, builds on our past work with the use of Delta-Enabled Grid Services (DEGS) [3, 17], which are Grid Services that have been extended with the capability of recording and externalizing incremental data changes using features such as Oracle Streams [16]. Whereas the work in [3, 17, 22, 23] 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 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. In particular, we outline the information that must be captured during process execution and communicated when failures occur to construct a complete picture of process dependencies. The decentralized approach eliminates the bottleneck and overhead reported in [3, 17] of forwarding all data changes to a central point for analysis. More importantly, the distributed delta object schedule and decentralized data dependency algorithm described in this paper represents 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, section II presents related work. Section III then outlines the functionality of PEXAs with an illustration of the decentralized data dependency analysis problem in section IV. Section V discusses the decentralized approach to the propagation of the recovery and graph construction process. The paper concludes in section VI with a summary and discussion of future research directions.

II. RELATED WORK

This section outlines related work. Section A first summarizes past work with advanced transaction models and transactional workflows, as well as recent work with recovery procedures for service-oriented environments. Section B then presents background on the DeltaGrid project that provides the basis for the work described in this paper.

A. Advanced Transaction Models and Transactional Workflows

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. In the work of [6], a mechanism was proposed to structure a long running process as a Saga. 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 [12]. In fact, current standards for web services, such as WS-BPEL [1] and WS-Business Activity [4] 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.

The term Transactional Workflows was introduced to recognize the relevance of transactions to workflow activity. Transactional 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 [21]. The ConTract Model provides a classic example of work with transactional workflows [19], 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 [5], the Crew Project [8], and METEOR [19]. 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.

Numerous other techniques are being investigated for addressing data consistency in service composition. Tentative holding is used in [9] to achieve a tentative commit state for transactions over Web Services. Acceptable Termination States (ATS) [2] are used to ensure user-defined failure atomicity of composite services. A reservation-based protocol is defined in [26], where a process uses an explicit reservation phase to request resources, followed by an explicit confirmation/cancellation phase. The concept of a promise in [7] is similar to the work in [26], 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. Other techniques include Web Services Composition Action [14, 15], WebTransact [11], and the work of [18], 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.

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.

B. The DeltaGrid Project

The research described in this paper builds on our past work with the DeltaGrid project [22, 23, 24] and Delta-Enabled Grid Services (DEGS) [3, 17]. 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 [3, 17], we experimented

with triggers and with the use of Oracle Streams as a way to capture data changes [16]. 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) [22, 23] 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* creates a log file that is used to support recovery activities when process execution fails [22, 23].

In particular, the global delta object schedule can be used to support the backward recovery of a completed service using *Delta-Enabled rollback* (DE-rollback). The delta schedule also provides the basis for discovering data dependencies among processes. As defined in [22, 23], a *process-level write dependency* exists if a process p_i writes an object *x* that has been written by another process p_j before p_j completes ($i \neq j$). An *operation-level write dependency* exists if an operation op_{ik} of process p_i writes an object that has been written by another operation op_{jl} of process p_j. Operation-level write dependency can exist between two operations within the same process ($i = j$). The operations that are write dependent on a specific operation op_{jl} form op_{jl}'s *write dependent set*. If op_{ik} is write dependent on op_{jl}, the enclosing process of op_{ik} is also write dependent on op_{jl}. Note that DE-rollback of an operation is only performed if the operation's write dependent set is empty.

Similar definitions exist to define read dependencies, but since a DEGS does not capture read information, the global execution context can be used to reveal *potential read dependency* among operations. An operation op_{ik} is potentially read dependent on another operation op_{jl} if: 1) op_{ik} and op_{jl} execute on the same DEGS, and 2) the execution duration of op_{ik} and op_{jl} overlaps, or op_{ik} is invoked after the termination of op_{jl}. The operations that are potentially read dependent on an operation op_{jl} form a set referred to as op_{jl}'s *read dependent set*. Potential read dependency can be defined at the process or operation levels.

An object interface is used to query the delta object schedule to return information about read and write dependencies. This information identifies concurrently executing processes that may be affected by the failure and recovery of a process that is accessing shared data. A user-defined rule-based approach for recovery actions of processes that are dependent on a failed process is addressed in [22, 25].

The research in [3, 17, 22, 23, 24] 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.

III. PROCESS EXECUTION AGENTS (PEXAS)

This section provides an initial overview of process execution agents. The discussion begins with an example execution scenario in Fig. 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 (Di) that has a DEGS interface and possibly 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. Section A presents an example execution scenario. Section B then describes the internal architecture of a PEXA.

A. A PEXA Execution Scenario

As shown in Fig. 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_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 Fig. 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, Y1, and Z1 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.

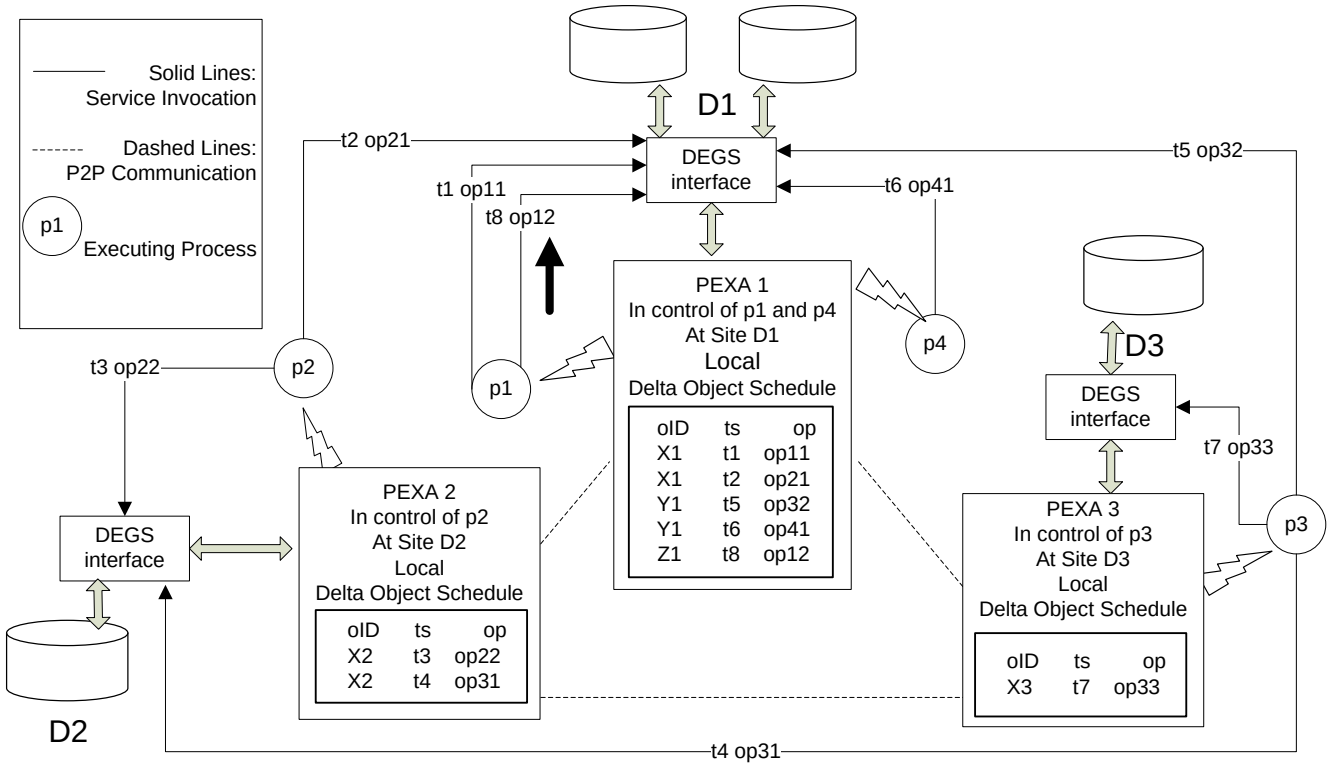

Figure 1. Decentralized Process Execution Agents

B. *Internal PEXA Architecture*

Fig. 2 shows the internal architecture of a PEXA. A PEXA contains a process execution component with a Process History Capture System that records runtime information about the status of each executing process. In the current environment, we are simulating the process execution component. Our current implementation uses the db4o object-oriented database [10] 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. Our future work will integrate the use of BPEL [1] into the PEXA architecture.

The local delta object schedule is an indexing structure defined in [22] 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 [22, 23]. Dependent processes can then query delta values, checking user-defined conditions to determine if they need to recover (e.g., 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. As a result, a PEXA also contains a peer-to-peer communication component that uses the JXTA message exchange protocol from Sun Microsystems [20].

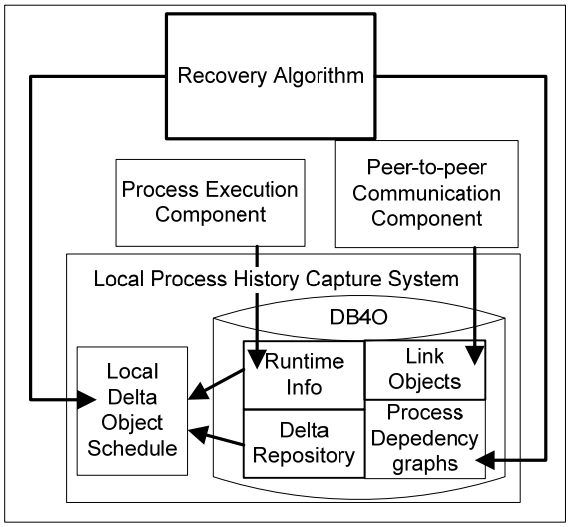

Figure 2. Internal PEXA Architecture

Furthermore, local process dependency graphs are extended with a structure known as a *link objec*t to assist in the construction of the global, distributed view. Section IV elaborates on the use of link objects and other runtime information to construct global, distributed process dependency graphs.

IV. 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 at this stage in the research, we assume 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. We use this as a worst-case scenario for constructing the full process dependency graph. We will address extensions to this simplification at the end of the paper when we discuss 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 [22, 23], the construction of a global process dependency graph is straightforward. The challenge with 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 Fig. 1. Fig. 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 portion of Fig. 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. Fig. 3 and Fig. 4 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 graph for p_1 due to the distributed nature of the execution. As shown in Fig. 3, 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. We refer to disconnected graphs such as those in PEXA 1 of Fig. 4 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.

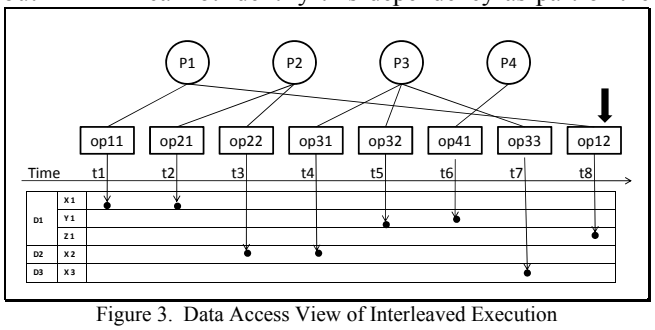

Figure 3. Data Access View of Interleaved Execution

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 was 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 Fig. 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 Fig. 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 Fig. 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. Section V elaborates on the algorithm for constructing distributed process dependency graphs among decentralized PEXAs.

V. Distributed Graph Construction and Recovery Process

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. Recall that 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. In the following sections, we first address process dependency graph construction. In the interest of space, we do not present the details of the

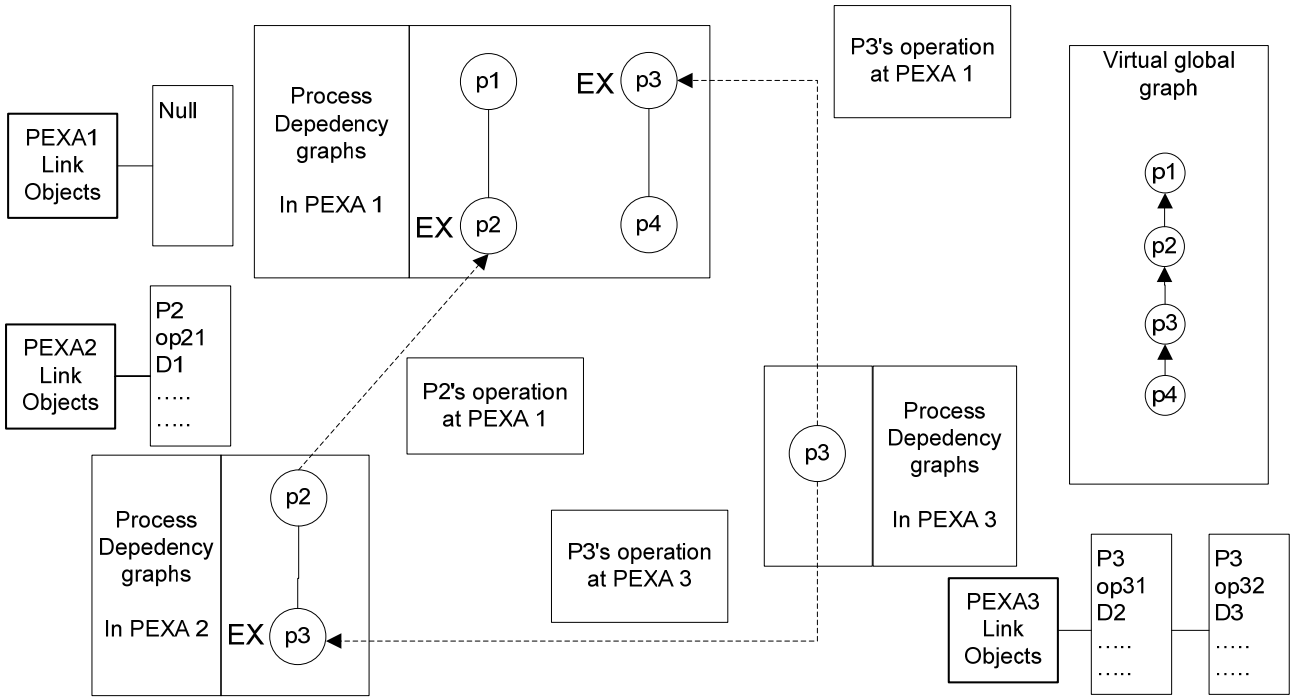

Figure 4. Global, Distributed Process Dependency Graph

findProcessDependencies(processId) algorithm, but outline relevant issues related to graph construction. We then address the use of external processes and link objects and demonstrate propagation of the recovery and graph construction process using the execution scenario introduced in Fig. 1.

A. Preliminary Issues for Graph Construction and Analysis

Information about a service execution that was requested by an external process is stored in the runtime 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)
- inOrEX (indicating internal or external process)
- degsId (DEGS identifier)
- PEXAId (the controlling PEXA identifier)
- status (the execution status of the process)

The inOrEx field distinguishes between service execution requested by a local process and service execution requested by an external process of 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.

Link objects are created by a PEXA when a process executing at the 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)
- PEXAId (PEXA identifier)
- status (indicating successful or compensated)

Link objects are also needed for propagation of the recovery and graph construction process.

A process dependency graph is created by findProcessDependencies(processId) at the process execution level rather than at the service execution 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 dependencies. The graph is represented as a hashmap 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. Read and write dependencies are found using procedures in [22] for querying a delta object schedule together with the process execution context. 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: $p_1 \leftarrow p_2 \leftarrow p_3 \leftarrow p_1$, where p_1 and p_3 are dependent on each other, but the dependency of p_3 on p_1 was created before the dependency of p_1 on p_3. Since the graph is constructed to control the order of the recovery process, cyclic information 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. 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 g.addVertex(p_i) is used to add nodes that represent processes (p_i) to the graph (g). The method g.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 g.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 link object attribute inOrEx 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 will be illustrated in the following two subsections.

B. Recovery and Graph Propagation

Fig. 5 provides pseudocode for the recovery and graph propagation process. The procedure is called after the construction of the local process dependency graph and is passed an ordered list of processes to be recovered. The list is created by doing a breadth-first traversal of the local process dependency graph.

The recover procedure examines each process in the list, finds operations of the process that were executed locally, and invokes compensating procedures for each process. For internal processes (i.e., the IF part of the algorithm), the algorithm then queries the link objects associated with the process to find services of the process that were executed at other sites. Notifications are then sent to the PEXAs of each external process. Each PEXA that is notified will invoke findProcessDependencies(processId) for the relevant process to construct its own local dependency graph to continue the recovery process at the new PEXA site. When receiving the notification, the process will be checked to see whether it is already in the graph. If it exists, the notification is ignored.

For a service invoked by an external process (i.e., the ELSE part of the algorithm), a notification is sent to the external PEXA to propagate the recovery and graph-building process. The notification triggers the graph construction in another site if the process from the notification is not in the local graph, and then compensates processes in the graph.

C. Execution Scenario

Fig. 6 uses the execution scenario from Fig. 1 to illustrate the logic of the algorithm presented in Fig. 5. 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 Fig. 6. Notifications that are initiated by the sendNotification procedure in Fig. 5 are drawn as dashed lines in Fig. 6.

```
// 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 processId in the list
   {
      //find operations from the local 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)
            {
                  For each operationId in the tempList
                  {
                     //mark compensated operations
                     LinkObject.updateOperation(operationId,
                                                "compensated");
                     String pexaId=
                                 LinkObject.getExecutingPexa(operationId);
                     sendNotification(processId, operationId, pexaId);
                  }
             }
      ELSE //the process is initiated by a peer PEXA
      {
            //mark compensated process
            For each operationId in the tempList
                OperationInfo.updateOperation(operationId, "compensated");
            //send notification
            String pexaId=
                  ProcessInfoAccess.getProcessInfo(processId).getPEXA();
            sendNotification(processId, pexaId);
      }
   }END FOR
}
```

Figure 5. The recover Procedure

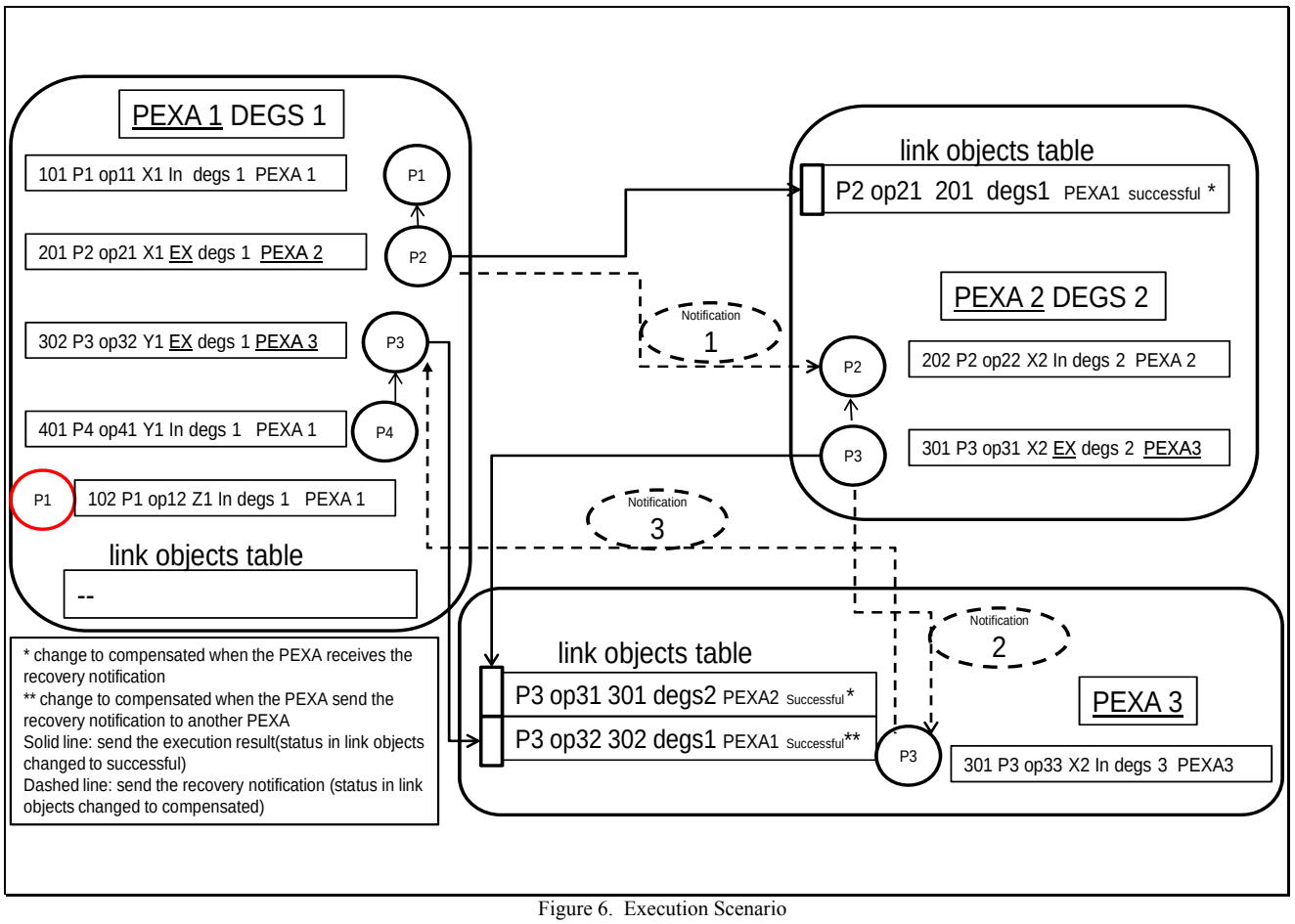

Figure 6. Execution Scenario

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 Fig. 6, 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 D1. 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 Fig. 6), 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 Fig. 5 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 Fig. 6, 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 Fig. 6) 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 Fig. 5 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 Fig. 6). PEXA 3 will be able to determine 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.

VI. CONCLUSIONS

This paper has provided an overview of our work with Process Execution Agents for decentralized data dependency analysis among concurrently executing processes in a service-oriented environment. PEXAs monitor the execution of processes and support the dynamic discovery of data dependencies that can be used to enhance recovery procedures by identifying processes that may be affected by shared access to the data of failed procedures. We have implemented the procedures described in this paper and are currently developing a simulation environment for the algorithms to conduct performance studies and to address scalability issues. The approach described in this paper represents a new way of integrating existing transaction processing concepts 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.

The algorithm presented in this paper represents a lazy approach to dependency analysis since the algorithm is not invoked until a process fails. We are also developing an eager approach to data dependency analysis, where the dependency graph is constructed during process execution and readily available when a failure occurs. Another important issue for future research includes the investigation of the fault tolerance of the data dependency algorithm in the context of distributed communication failures.

In this initial stage of the research, we have assumed that a failed process and its dependent processes implement compensation as a recovery procedure. Future directions are focused on integrating the decentralized data dependency analysis algorithms with the service composition model that in [24], integrating the PEXA concept with BPEL engines, and addressing the impact of the approach on process consistency. We are enhancing the model with formal methods for user-defined specification of correctness conditions and are investigating the manner in which these specifications can be used with PEXAs to develop more intelligent service-oriented execution environments for addressing data consistency issues for concurrent processes.

ACKNOWLEDGMENT

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

[1] "BPEL4WS V1.1 specification," URL http://www.ibm.com/developerworks/library/specification/ws-bpel/, 2003.

[2] S. Bhiri, O. Perrin, and C. Godart, "Ensuring required failure atomicity of composite web services," in Proc. of the 14th Int. Conference on World Wide Web, 2005, pp. 138-147.

[3] L. Blake, "The Design and Implementation of Delta-enabled Grid Services," M.S. Thesis, Arizona State University, 2006.

[4] L. F. Cabrera, G. Copeland, T. Freund et al., "Web Services Business Activity Framework (WS-BasinessActivity) ," ftp://www6.software.ibm.com/software/developer/library, WS-BusinessActivity.pdf, 2005.

[5] J. Eder and W. Liebhart, "The workflow activity model WAMO," in Proc. of the 3rd Int. Conference on Cooperative Information Systems (CoopIs), 1995.

[6] H. Garcia-Molina and K. Salem, "Sagas," in Proc. of the ACM SIGMOD Annual Conference on Management of Data, 1987, pp. 249-259.

[7] P. Greenfield, A. Fekete, J. Jang et al., "Compensation is not enough," in 7th Int. Conference on Enterprise Distributed Object Computing, 2003.

[8] M. Kamath and K. Ramamritham, "Failure handling and coordinated execution of concurrent workflows," in Proc. of the IEEE Int. Conference on Data Engineering, 1998.

[9] B. Limthanmaphon and Y. Zhang, "Web service composition transaction management," in Proc. of the 15th Australasian Database Conference, 2004, pp. 171-179.

[10] J. Paterson, S. Edlich, H. Hörning et al., The Definitive Guide to db4o: Apress Berkely, CA, USA, 2006.

[11] P. F. Pires, M. R. F. Benevides, and M. Mattoso, "Building reliable web services compositions," Lecture Notes in Computer Science, pp. 59-72, 2003.

[12] A. Rolf, W. Klas, and J. Veijalainen, Transaction management support for cooperative applications: Kluwer Academic Publishers, 1998.

[13] M. P. Singh and M. N. Huhns, Service-oriented computing: semantics, processes, agents: Wiley, 2005.

[14] F. Tartanoglu, V. Issarny, A. Romanovsky, and Nicole Levy, "Coordinated forward error recovery for composite web services," in Proc. of the IEEE Symposium on Reliable Distributed Systems, 2003, pp. 167-176.

[15] F. Tartanoglu, V. Issarny, A. Romanovsky, Nicole Levy, "Dependability in the Web services architecture," Lecture Notes in Computer Science, pp. 90-109, 2002.

[16] M. Tumma, Oracle Streams: High Speed Replication and Data Sharing: Rampant TechPress, 2004.

[17] S. D. Urban, Y. Xiao, L. Blake, and S. W. Dietrich, "Monitoring Data Dependencies in Concurrent Process Execution through Delta-

Enabled Grid Services," International Journal of Web and Grid Services, vol. 5, no. 1, 2009, pp. 85-106.

[18] K. Vidyasankar and G. Vossen, "A multilevel model for Web service composition," in Proc. of the IEEE Int. Conference on Web Services, 2004, pp. 462-469.

[19] H. Wächter and A. Reuter, The contract model: Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1992.

[20] B. J. Wilson, Jxta: New Riders Boston, 2002.

[21] D. Worah and A. Sheth, "Transactions in transactional workflows," Advanced Transaction Models and Architectures, pp. 3-34, 1997.

[22] Y. Xiao, "Using deltas to analyze data dependencies and semantic correctness in the recovery of concurrent process execution," PhD Dissertation, Arizona State University, 2006.

[23] Y. Xiao and S. D. Urban, "Process Dependencies and Process Interference Rules for Analyzing the Impact of Failure in a Service Composition Environment," Journal of Information Science and Technology, vol. 5, no. 2, 2008, pp. 21-45.

[24] Y. Xiao and S. D. Urban, "The DeltaGrid Service Composition and Recovery Model," International Journal of Web Services Research, vol. 6, no. 3 2009.

[25] Y. Xiao and S. D. Urban, “Using Data Dependencies to Support the Recovery of Concurrent Processes in a Service Composition Environment,” Proc. Of the Cooperative Information Systems Conference, Monterrey, Mexico, Nov. 2008, pp. 139-156.

[26] W. Zhao, F. Kart, L. E. Moser, and P. M. Melliar-Smith, “A Reservation-Based Extended Transaction Protocol for Coordination of Web Services,” International Journal of Web Services Research, vol. 19, no. 2, 2008, pp. 188-203.