Handling Faults in Decentralized Orchestration of Composite Web Services

Girish Chafle, Sunil Chandra, Pankaj Kankar, and Vijay Mann

IBM India Research Laboratory, New Delhi, India
{cgirish, csunil, kpankaj, vijamann}@in.ibm.com

Abstract. Composite web services can be orchestrated in a decentralized manner by breaking down the original service specification into a set of partitions and executing them on a distributed infrastructure. The infrastructure consists of multiple service engines communicating with each other over asynchronous messaging. Decentralized orchestration yields performance benefits by exploiting concurrency and reducing the data on the network. Further, decentralized orchestration may be necessary to orchestrate certain composite web services due to privacy and data flow constraints. However, decentralized orchestration also results in additional complexity due to absence of a centralized global state, and overlapping or different life cycles of the various partitions. This makes handling of faults arising from composite service partitions or from the failure of component web services, a challenging task.

In this paper we propose a mechanism for handling faults in decentralized orchestration of composite web services. The mechanism includes a strategy for placement of fault handlers and compensation handlers, and schemes for fault propagation and fault recovery. The mechanism is designed to maintain the semantics of the original specification while ensuring minimal overheads.

1 Introduction

A composite web service is created by aggregating the functionality of existing web services (which act as its *components*) and can be specified using XML based languages like BPEL4WS [1], WSIPL [9], WSCI [3], *etc.* Typically, a composite service is *orchestrated* by an orchestrator node in a *centralized* manner. The orchestrator node receives the client requests, invokes the component web services and makes the required data transformations as per the service specification. We refer to this mode of execution as ***centralized*** orchestration. In this mode, all data is transferred between the various components via the orchestrator node instead of being transferred directly from the point of generation to the point of consumption. This leads to unnecessary traffic on the network resulting in poor scalability and performance degradation at high loads. Furthermore, centralized orchestration may not be feasible in scenarios where component web services place constraints on access to the data they provide or on the source from which they can accept data.

B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 410–423, 2005.

We have been investigating decentralized orchestration in our prior work [6,7,15,16] in order to overcome the above mentioned limitations imposed by centralized orchestration. In ***decentralized*** orchestration [15], the composite web service specification is analyzed for data and control dependencies, and broken down into a semantically-equivalent set of partitions known as *topology*. The partitions execute independently without any centralized control and interact with each other directly by transferring data using asynchronous messaging. Our decentralization algorithm [15] reduces the data on the network by sending it directly from its point of generation to point of consumption.

The benefits of decentralization come with the added complexity of the system as decentralization involves partitions which execute independently and interact with each other directly using asynchronous messaging. The global state of the original composite service is now distributed across different partitions. A fault occurring in one partition does not get noticed by other partitions or the client issuing the request. Fault propagation (even in absence of fault recovery) becomes essential so that - a) a fault occurring in one partition does not lead to any other partition waiting indefinitely for an input from the erroneous partition , and b) a client issuing a request is notified about the fault occurring in a partition. Absence of fault propagation in decentralized orchestration will lead to degradation of system performance with increasing load as resources get held up. This is hardly an issue in centralized orchestration as faults are generated locally on the centralized node and the client can be notified easily. Fault recovery, on the other hand, is required in order to correct the effects of partial changes to the state of the system and restore the system to an error free state.

A mechanism for fault handling, has to be designed such that it does not degrade the system performance under normal execution and at the same time be as efficient as possible in case of a fault. However, designing such a mechanism for decentralized orchestration is non trivial because of the following challenges:

- In contrast to centralized orchestration, there is no centralized global state as different partitions execute on different nodes.
- When a composite service specification is partitioned, the fault and compensation handlers have to be placed appropriately amongst the partitions in order to maintain correct semantics of the original service specification. Furthermore, the partitions need to be augmented with additional code to correctly forward and handle faults.
- Composite service languages such as BPEL4WS define "scope" activity to associate fault handlers and compensation handlers with a fault handling and recovery context. However, in decentralized orchestration, a single scope might get partitioned across various partitions and the partitioned scopes might execute at different times and have either overlapping or different lifecycles. No single context exists that can store the data of already completed scopes which is needed to compensate them while recovering from a fault.

In this paper we propose a mechanism for handling faults in decentralized orchestration of composite web services.The mechanism includes a strategy for placement of fault and compensation handlers and schemes for fault propagation

and fault recovery. The mechanism is designed to maintain the semantics of the original service specification while ensuring minimal overheads.

2 Background and Related Work

Lot of work has been done in the area of fault handling in distributed systems. Two types of approaches have been proposed for fault recovery - *backward error recovery* and *forward error recovery* [14]. Forward error recovery is based on the use of redundant data that repairs the system by analyzing the detected fault and putting the system into a correct state. In contrast, backward error recovery returns the system to a previous (presumed to be) fault-free state without requiring detailed knowledge of the faults.

Various fault handling models for flat and nested workflow transaction hierarchies have also been proposed in literature [8,18]. BPEL4WS uses a fault handling model that supports nested transactions and allows inner transactions (or sub-transactions) to make their results visible externally (referred to as open nested transactions [18]). In this paper we adhere to the fault handling model used by BPEL4WS and propose mechanisms that help conforming to the BPEL4WS fault handling model during decentralized orchestration of BPEL4WS composite services.

Application partitioning systems (JOrchestra [17], Coign [11], etc) that replace local method calls by remote method calls make use of various forms of forward or backward error recovery scheme as the control for the application remains centralized and there exists a single context in which faults are handled.

Various workflow systems (including systems that employ workflow partitioning) relied heavily on backward error recovery, (although forward error recovery can also be used here) as most of the underlying resources were usually under the control of a single domain. These are specified using proprietary languages and usually do not handle nested scopes [10,13].

Fault recovery becomes a little more complex for composite web services as component web services may be distributed across different autonomous domains. Transactions (which fall under backward error recovery mechanisms), which have been successfully used in providing fault tolerance to distributed systems, are not suited in such cases because of following reasons:

- Management of transactions that span across web services deployed on different domains requires cooperation among the transactional systems of individual domains. These transactional systems may not be compliant with each other.
- Locking resources until the termination of the embedding transaction is in general not appropriate for such web services, still due to their autonomy, and also to the fact that they potentially have a large number of concurrent clients that will not stand extensive delays.

Forward error recovery is extensively used in composite web services in order to handle errors. For instance, BPEL4WS provides support for fault handling

to recover from expected as well as unexpected faults, and compensation to "undo" already committed steps by providing user defined fault handler and compensation handler constructs. Apart from the compensation handlers in BPEL4WS, lot of other efforts are also underway for providing transactional support for composite web services (BTP [2], and WS Transaction [4]). Other related work includes Web Service Composition Action (WSCA) [12]. WSCA is an extension of the Coordinated Atomic Action (CA Action) for web services, and is used for handling concurrent exceptions.

Decentralized orchestration of composite web services further complicates fault handling as discussed earlier. In addition to augmenting the existing forward error recovery mechanisms, additional fault propagation and data collection schemes are needed. Not much work has been done in this area.

3 Proposed Fault Handling Mechanism

As discussed in section 2, forward error recovery mechanisms are used for composite web services. The proposed mechanism is also based on the same. It is designed to maintain the semantics of original centralized specification and handles only those faults which are handled either explicitly or implicitly in the original centralized specification. The overall mechanism is based on correct partitioning of the input service specification, such that the fault handlers and compensation handlers are placed appropriately amongst the different partitions, and on augmenting the partitions with additional code that aids in fault propagation, data collection and fault recovery. The fault handling mechanism consists of three runtime phases when a fault occurs. The first phase ensures the propagation of a fault to a partition where its corresponding fault handler resides. The second phase consists of collecting the data, that is needed to initiate fault recovery, from different partitions. The third phase consists of handling the fault in the appropriate fault handler and initiating forward recovery by invoking compensation handlers of already completed inner scope(s). These phases are described in detail in the following subsections with the help of BPEL4WS constructs.

3.1 Placement of Fault Handlers and Compensation Handlers

BPEL4WS specification defines "scope" activities to associate fault handlers and compensation handlers with a fault handling and recovery context . Every BPEL4WS process has an implicit scope of its own. An explicit "scope" activity or an implicit scope can consist of many activities, each of which could itself be a "scope" activity, thereby resulting in *nested scopes*. Since, any activity may generate a fault, the control may flow from any activity to the fault handlers. Further, a fault handler may completely handle the fault or may re-throw it to the outer scope. If a fault handler handles a fault, the execution in the outer scope resumes normally. However, any scope, whose associated fault handler has been invoked, is not considered to be completed normally and compensation is not enabled for it and can not be called on it from an outer scope. Compensation is

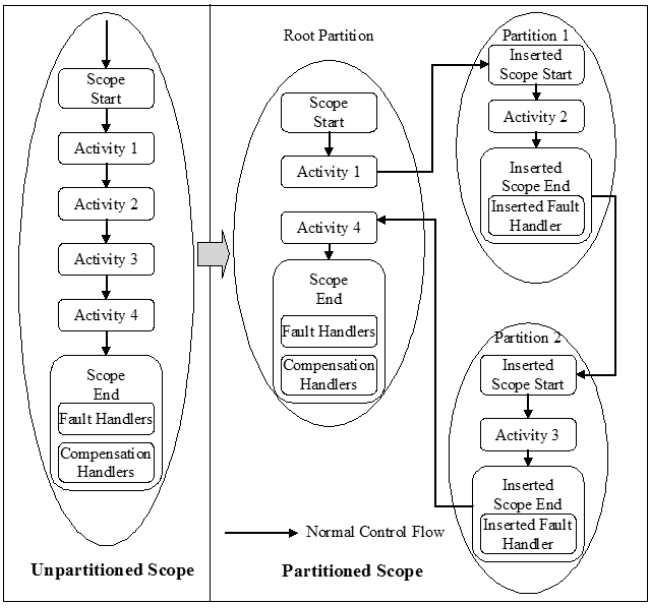

Fig. 1. Partitioning of scope and placement of fault and compensation handlers

enabled only for those scopes that have completed normally and a compensation handler is always invoked from a fault handler or a compensation handler of the outer scope.

In decentralized orchestration, activities inside a scope may get arbitrarily partitioned and thus placement of fault and compensation handlers requires special attention. The overall decentralization algorithm presented in [15] works as follows. In the proposed solution, the decentralization algorithm partitions a scope in such a manner that the start and end of each scope reside in the same partition (which is referred to as the **root** partition of that scope) and the rest of the activities of the scope are placed as per the algorithm described in [15]. We anchor the fault handlers and compensation handlers to the end of scope (refer figure 1). This means that the fault handlers and compensation handlers for a scope always reside in the *root partition* of that scope. Theoretically, the end of each scope can reside on a partition that is the last partition in the control flow of a particular scope. Thus, the start and end can reside on different partitions. The end of the scope partition will then host the fault handlers and compensation handlers according to the scheme given in this paper. However, this will require creation of a logical "scope" that is different from the physical "scope" activity in BPEL4WS. Furthermore, state information will have to be transferred from the start of scope partition to the end of scope partition. To avoid this complexity, the start and end of each scope are placed together. In case of conditional activities like `while` and `switch`, the root partition is the partition that contains the condition itself.

The algorithm given in [15] has been modified to ensure that the fault handlers and the compensation handlers are anchored to the root partition of a scope. We prepare the *Control Flow Graph (CFG)* preserving the fault handlers and compensation handler. A fault handler may have various *Catch* blocks for handling different faults and a *CatchAll* block for handling any fault. A *control flow* edge is added from every activity to the first activity under *CatchAll* since control can flow from any activity to the fault handlers. For all the other fault handlers, a separate control flow edge is added from all those activities which can throw the fault handled by this handler to the first activity of the handler. Similarly, a control flow edge is added from the last activity of the scope to the start activity of the compensation handler because as per BPEL4WS specification, the compensation handler, if invoked, will see a frozen snapshot of all variables, as they were when the scope being compensated was completed. After that we run the *Reaching Definitions* algorithm [5] to generate all the *data flow* edges to discover data dependencies between activities and corresponding activities of the fault and compensation handlers. These data flow edges are used to determine what data needs to be propagated to the root partition during the data collection phase, which is subsequently used during fault handling and recovery.

The data flow edges getting in or out of the fault handlers and compensation handler are marked as fault edges as these are different from the normal data flow edges between activities of the scope. Before we construct the *Control Dependent Graph (CDG)* we cut the sub-trees corresponding to fault handlers and compensation handler from their scopes. After this the *Merge* algorithm (given in [15]) runs on the modified CDG in the normal fashion, ignoring the fault data flow edges. Since, the sub-trees corresponding to fault handlers and compensation handler are absent from the CDG given to *Merge* algorithm, the algorithm will not create partitions for the activities of these handlers. After creating all the partitions, these handlers are added to the their respective scopes during code generation, thereby ensuring that the fault handlers and compensation handlers reside in the root partition.

3.2 Fault Propagation

Since a fault may occur in any partition and the corresponding fault handler resides in the *root partition*, the fault needs to be propagated to the root partition. A partition forwards a fault, that has either occurred within that partition or it has been received from another partition as part of the fault propagation scheme, to one of the following partitions (at the same level in PDG [15]), whichever comes first, in the control flow path

- Root partition of the given scope: All the fault handlers and compensation handlers associated with a scope reside in the root partition of the given scope (shown in figure 1) and all faults occurring within a scope are eventually routed to the root partition (shown in figure 2).
- Next join partition (a partition having more than one incoming links): If there is a join partition (shown in figure 2) in between the given partition

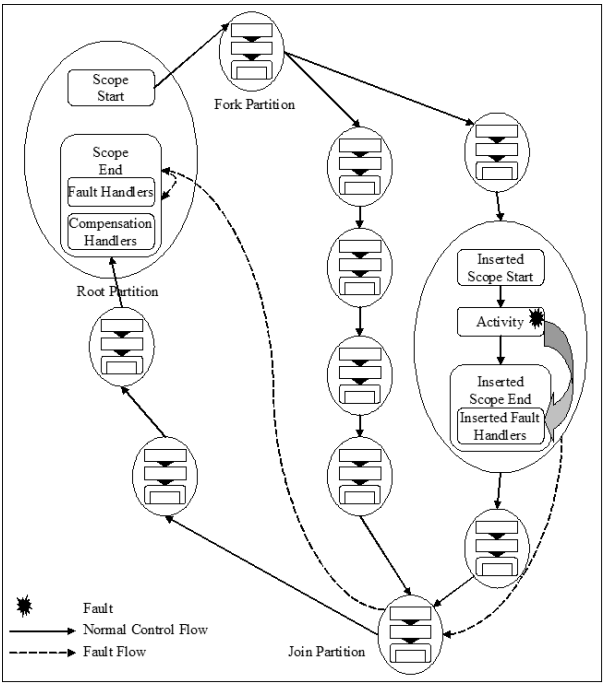

Fig. 2. Fault Propagation Scheme

and the root partition, then the fault is sent to the join partition because a join partition expects input messages from all the incoming links (which execute in parallel legs) and it will continue to wait forever for a message from the leg in which a fault has occurred in one of the partitions. To prevent this condition, the fault is sent to the join partition. The join partition, in turn, upon receiving all the incoming messages (including the fault), forwards the fault to the next partition as per the fault propagation algorithm.

– Next fork partition (a partition having more than one outgoing links): If there is a fork partition (shown in figure 2) in between the given partition and the root partition, then the fault is passed to the fork partition which then forwards the fault on all the outgoing links according to the fault propagation algorithm. A fork partition results in more than one outgoing links and these links may join at some other partition (which becomes a join partition) in the control flow. This join partition will expect an input message from all its incoming links, some of which are the outgoing links of the fork partition. Therefore a fork partition needs to forward a fault on all its outgoing links.

During code generation, all partitions except *root partition* of the process scope, are inserted with additional fault handlers (shown as inserted fault handler in figure 2) that help in fault propagation. These fault handlers are inserted in the outermost scope (which is there implicitly as all partitions consist of a

BPEL4WS `process` activity and a `process` has an implicit scope of its own) of each partition. One handler (a BPEL4WS *Catch* clause) is inserted for each type of fault handled in the original specification. If the original specification does not have a `CatchAll` fault handler, one such handler is inserted to catch other unknown faults which may be generated and need to be propagated to the root of the scope. These fault handlers pack the fault name and fault data associated with the handler and forward it. The fault handlers then wait for a control message to forward data (a snapshot of BPEL4WS variables) required for fault recovery. The `receive` activities in all partitions, to which a fault can be propagated (*i.e.,* root partitions of all scopes, fork partitions and join partitions), are replaced by `pick` activities during code generation. A `pick` activity allows reception of one of n possible incoming messages. Out of these *n* possible incoming messages, one is an actual input message and the rest *n-1* incoming messages correspond to the *n-1* fault messages which a partition can receive due to *n-1* types of faults handled in that scope in the original specification. In order to receive these *n* possible messages, a new `operation` is added to the same `port type` (which existed for the actual incoming message) as part of a new `OnMessage` activity inside `pick`, for each of the *n* possible messages. Upon receiving a fault message, a partition throws a fault (representing the input fault message), which is then caught by the inserted fault handlers in the inserted implicit scope in case of *fork partitions* and *join partitions*, which then propagate the fault according to the fault propagation scheme. In case of a *root partition* the fault is caught and handled by the actual fault handlers associated with the given scope. If a fault handler re-throws a fault, it is automatically caught by the *inserted fault handler* that is associated with the implicit outer scope of the partition in which the inner scope resides. The inserted fault handler is augmented with code to propagate the fault to the root partition of the outer scope using the fault propagation scheme explained above.

3.3 Data Collection

The second phase of the proposed mechanism consists of collecting the data that is required to recover from a fault. All partitions that complete successfully, wait for a control message. If the composite service completes successfully, the *root partition* of the top level scope(*i.e.,* the client facing partition) sends a `NormalComplete` control message along the path traversed by request. All the partitions exit normally on the receipt of this message. In case of a fault, the fault handler of the *root partition* of the scope in which the fault occurred, sends a `DataCollection` control message to its next partition(s) according to the control flow (see figure 3). The message flows along the path traversed by the request till it reaches the partition where the fault occurred. From there, it flows along the path traversed by the fault (as per the fault propagation scheme). Each partition upon receiving the *DataCollection* control message, appends its variables *i.e.,* the data to the payload of the message as a message part. The data that needs to be appended is determined using the modified decentralization algorithm given in section 3.1. The *root partitions* of all the scopes except the top level scope for

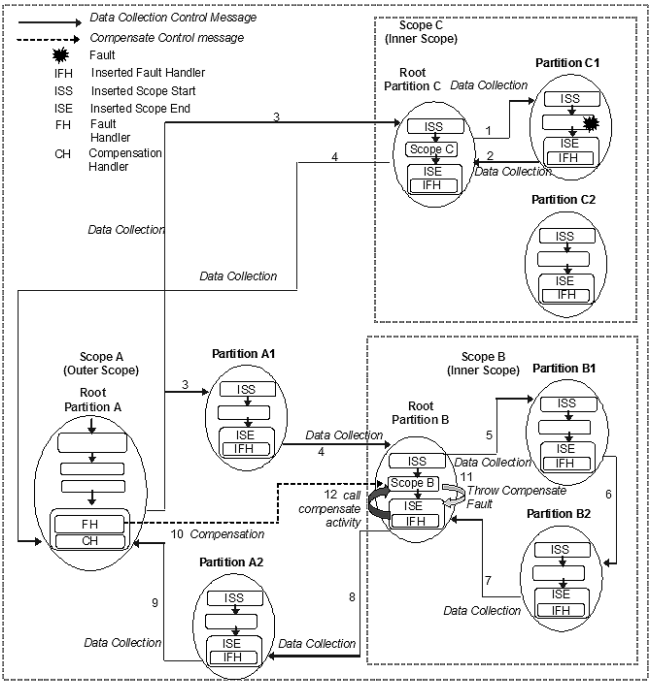

Fig. 3. Flow of control messages in case of a fault

the composite service, now enter a state where they wait for a `Compensation` or `NoCompensation` control message from their outer scope.

In implementation, a set of extra activities are added at the end of each partition. This includes a `pick` to receive either a `DataCollection` or a `NormalComplete` control message from previous partitions, a set of `assign` activities to pack the data to be sent for fault handling and compensation and a set of `invoke(s)` to send the control messages to next partitions in the control flow path. A similar set of activities are also added at the end of the inserted fault handlers (associated with the inserted implicit scope) in all the partitions. The only difference is that instead of a `pick`, a `receive` activity is added as it will never receive a `NormalComplete` control message and can receive only `DataCollection` control message, as a fault has already occurred in that scope.

3.4 Fault Recovery

Once the fault is propagated to the *root partition* of a scope, the corresponding fault handler, if provided, is triggered. If no fault handler is provided, default semantics are provided by the BPEL4WS engine hosting the *root partition*. The data collected during the second phase is used by the root partition for recovering from faults in its scope or faults thrown by its inner scopes. The fault handler

associated with the root partition first executes the activities that are specified in the original specification to handle the given fault.

Normally completed scopes may need to be compensated, if a fault occurs in the outer scope or if the outer scope is being compensated. If a fault or compensation handler is provided in the original service specification that consists of explicit `compensate` activities targeted at inner scopes, then these `compensate` activities are replaced with `invoke(s)` that send a `Compensation` control message to the *root partitions* of all such inner scopes. For all other inner scopes (which don't serve as targets for any explicit `compensate` activity), a `NoCompensation` control message is sent to their *root partitions*. If there is no fault handler or compensation handler provided in the original service specification, a `Compensation` message is sent to the *root partitions* of all the inner scopes in the reverse order of their occurrence. This is shown in figure 3, where the outer scope - Scope A, has two inner scopes - Scope B and Scope C, that execute in parallel. Scope B finishes successfully, while a fault occurs in Scope C, which is first propagated to the root partition of Scope C. The root partition of Scope C initiates a data

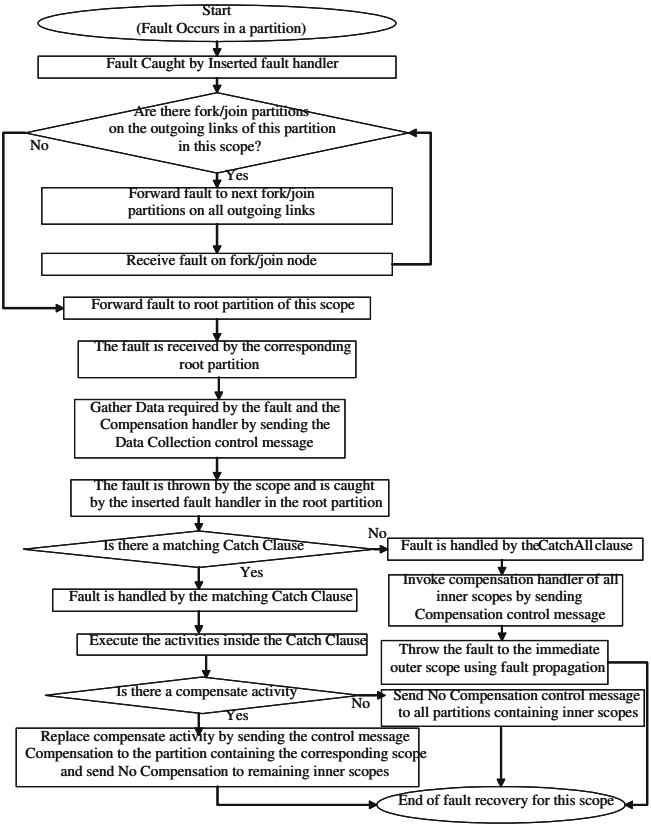

Fig. 4. Flowchart for fault propagation, data collection and fault recovery

collection phase for its scope and then throws the fault to the outer scope - Scope A, as there is no matching fault handler in Scope C. Scope A then initiates the data collection phase for Scope B and Scope C and once the data collection finishes, it compensates Scope B.

The `Compensation` (or the `NoCompensation`) control message is received by the *root partition* of inner scopes (refer figure 3). During decentralization, an additional fault handler is inserted in the top level implicit scope of each root partition, since compensation handler can be invoked only from a fault handler or a compensation handler of the immediately outer scope. When the root partition receives the `Compensation` message, it throws a specific "Compensation" fault which is caught by this inserted fault handler. This fault handler simply invokes the compensation handler of the inner scope by using explicit `compensate` activity as shown in figure 3.

Partitions containing `while` or `switch` blocks require further attention. The `switch` block maintains information about the `case` that was chosen, so that same leg is chosen for data collection and compensation phases. A `while` block can result in a number of iterations each leading to an instance of all the partitions that are part of that block. This raises issues related to the timing of data collection and fault recovery. We are working on augmenting our scheme with all these considerations for `while` loops.

All the code required for the mechanism is automatically generated by the *decentralization tool*, so that developer of the original composite service specification is not burdened with its complexity. The flowchart for the overall fault handling mechanism is shown in figure 4.

4 An Example Scenario

In this section we explain the proposed fault handling mechanism with the help of an example. To highlight all the important aspects of the scheme we created a composite service specification which includes a *switch*, an *inner scope* and few *fork* and *join* points. There are a total of 22 *invoke*s, a *switch* in the outer scope, and an inner scope. The arrangement of various activities in the BPEL4WS code leads to a decentralized topology that helps us in explaining the scheme in its full generality. We exclude the input BPEL4WS specification due to space constraints.

4.1 Fault Propagation

The fault propagation scheme is shown in action for the example scenario in figure 5. Four different requests resulting in four faults originating at different partitions are shown in the figure.

In the first request, fault $F1$ occurs in $P17$, which then forwards it to the next join partition $P13$. Partitions $P18$ (and the inner scope) and $P19$ are skipped and do not get instantiated. The other leg forked at $P12$ proceeds as usual and reaches the join partition $P13$. After receiving the two messages (the fault

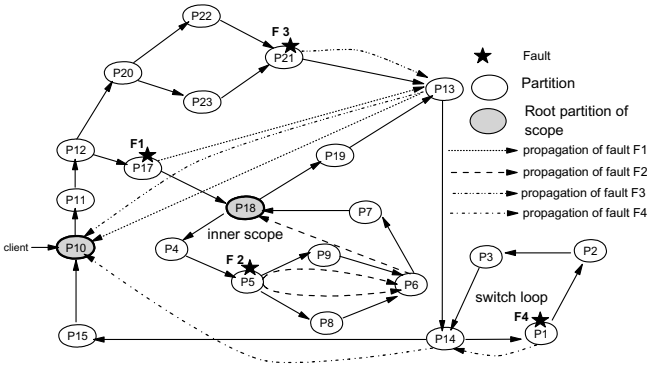

Fig. 5. Fault propagation in an example decentralized topology

message from $P17$ and a normal input message from $P21$), partition $P13$ sends the fault to the end of the scope, $P10$. Remaining partitions after $P13$ in the control flow are skipped and do not get instantiated for this request.

In the second request, fault $F2$ occurs in the inner scope at fork partition $P5$. $P5$ sends two fault messages to the next join partition $P6$, one for each outgoing leg. Partition $P6$ then sends the fault to the *root partition* of the inner scope, $P18$, where it is handled. Partitions $P7$, $P8$, and $P9$ of the inner scope are not instantiated for this particular request.

In the third request, fault $F3$ occurs in partition $P21$ which forwards it to the next join partition $P13$. The other leg forked at $P12$ including the inner scope completes normally. After receiving the two messages,(the fault message from $P21$ and a normal input message from $P19$), partition $P13$ sends the fault to the *root partition* the scope, $P10$. Remaining partitions in the control flow after $P13$ are skipped for this particular request.

In the fourth request, fault $F4$ occurs inside the `switch` block in the outer scope at $P1$, is sent to the root partition of the *switch* block $P14$, and is forwarded to the root partition of the outer scope, $P10$. Partitions $P2$, $P3$, and $P15$ are not instantiated for this particular request. The partitions preceding the switch block in control flow, get completed before the fault occurs and may need to be compensated by the compensation handlers present in the outer scope.

4.2 Data Collection

We demonstrate data collection phase for the second request which results in fault $F2$, occurring inside the inner scope and for the fourth request which results in fault $F4$, occurring inside the `switch` block, as explained above.

For the second request, the fault $F2$ reaches the *root partition* of the inner scope $P18$, which then sends the `DataCollection` control message to first partition ($P4$) in the control flow. The message then travels along the control flow path till partition $P5$ where the fault occurred. After that it follows the fault propagation path. On receiving the message, each partition appends its variables

i.e., the data that are required for executing the fault handlers and compensation handlers, to the payload of the message as a message part, and forward the message to the next partition and exit. The collected data finally reaches the root partition, $P18$ which invokes the fault handler for this scope.

For the fourth request, the fault $F4$, reaches the *root partition* $P10$ of the outer scope, which then sends the `DataCollection` control message to its first partition ($P11$) in the control flow. The message then travels along the normal control flow covering all partitions which were instantiated (all partitions except $P2$, $P3$, and $P15$) during the course of execution of this particular request. These partitions wait for the data collection message. On receiving the control message, each partition appends its variables *i.e.,* the data that are required for executing the fault handlers and compensation handlers, to the payload of the message as a message part, and forwards the message to the next partition and exits. When the *root partition* of the inner scope, $P18$, gets the `DataCollection` message, it collects the variables that are local to the inner scope as well as those belonging to the outer scope. The variables local to the inner scope are stored at $P18$ and those belonging to the outer scope are appended to the `DataCollection` message, which is then forwarded to partition $P19$. The collected data finally reaches the *root partition* of the outer scope $P10$. All the partitions except the *root partition* of inner scope $P18$ exit after forwarding the data collection message.

4.3 Fault Recovery

The third phase of the scheme consists of execution of the activities inside a fault handler for a given scope in which a fault has occurred and compensating completed inner scopes. It makes use of the data collected during the second phase. For fault $F2$, the root partition of the scope $P18$ executes the fault handlers and the fault is not re–thrown to the outer scope.

For fault $F4$, the completed inner scope (root partition $P18$) needs to be compensated while executing the fault handler associated with the *root partition* of the outer scope - $P10$. The *root partition* $P10$ sends a `Compensation` control message to partition $P18$ which executes the compensation handlers for the inner scope using the data collected during the second phase and exits.

5 Conclusions

In this paper we have proposed a mechanism for handling faults in decentralized orchestration of composite web services. We adhere to the fault handling model used by BPEL4WS which employs forward error recovery (through compensation) and supports open nested transactions. The mechanism is designed to maintain the semantics of the original service specification while ensuring minimal overheads. We have implemented the fault propagation part of our scheme in the decentralization tool that we have developed and are currently working on the implementation of the rest of the scheme. We did not touch upon performance or complexity analysis of the proposed mechanism in this paper and

plan to investigate it in the near future. The fault management scheme proposed in this paper is centralized as fault handlers and compensation handlers for a scope reside in the root partition. This has been done to keep the scheme simple as the complexities associated with a decentralized fault management scheme might not be worth its benefits. We are going to further explore decentralized fault management in near future.

References

1. Business Process Execution Language for Web Services Version 1.1. http://www.ibm.com/developerworks/library/ws-bpel/.
2. OASIS Business Transaction Protocol, Committee, Specification 1.0. http://www.oasis-open.org/business-transaction.
3. Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/wsci.
4. Web Services Transaction (WS-Transaction). http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/.
5. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
6. G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proceedings of WWW*, 2004.
7. G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Orchestrating Composite Web Services Under Data Flow Constraints. In *Proceedings of IEEE International Conference on Web Services*, Orlando, USA, July 2005.
8. Q. Chen and U. Dayal. Failure Handling for Transaction Hierarchies. In *Proceedings of International Conference on Data Engineering (ICDE)*, 1997.
9. David W. Cheung, Eric Lo, C. Y. Ng, and Thomas Lee. Web Services Oriented Data Processing and Integration. In *Proceedings of WWW*, 2003.
10. Dickson K.W. Chiu, Qing Li, and Kamalakar Karlapalem. ADOME-WFMS: Towards Cooperative Handling of Workflow Exceptions. *Lecture Notes in Computer Science*, (2677):271–288, 2001. Advances in exception handling techniques.
11. G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of OSDI*, February 1999.
12. V. Issarny, N. Levy, A. Romanovsky, and F. Tartanoglu. Coordinated Forward Error Recovery for Composite Web Services. In *Proceedings of SRDS*, 2003.
13. Mohan U. Kamath and Krithi Ramamritham. Pragmatic Issues in Coordinated Execution and Failure Handling of Workflow Control Architectures. Computer Science Technical Report 98-28, University of Massachusetts, August 1998.
14. P.A. Lee and T. Anderson. *Fault Tolerance Pinciples and Practice, volume 3 of Dependable Computing and Fault Tolerant Systems*. Springer-Verlag, 1990.
15. M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing Execution of Composite Web Services. In *Proceedings of OOPSLA*, 2004.
16. M. G. Nanda and N. Karnik. Synchronization Analysis for Decentralizing Composite Web Services. In *Proceedings of SAC*, 2003.
17. E. Tilevich and Y. Smaragdakis. J-Orchestra - Automatic Java Application Partitioning. In *Proceedings of ECOOP 2002*, June 2002.
18. G. Weikum and H. Schek. Concepts and applications of multilevel transactions and open nested transactions. *Transaction Models for Advanced Database Applications*, 1992.