Synchronization Analysis for Decentralizing Composite Web Services

Mangala Gowri Nanda
IBM India Research Laboratory
Block 1, IIT, Hauz Khas
New Delhi 110016, India
mgowri@in.ibm.com

Neeran Karnik
IBM India Research Laboratory
Block 1, IIT, Hauz Khas
New Delhi 110016, India
kneeran@in.ibm.com

ABSTRACT

Web Services are emerging as the standard mechanism for making information and software available programmatically via the Internet, and as *building blocks* for applications. A *composite* web service may be built using multiple component web services. Once its specification has been developed, the composite service may be *orchestrated* either using a *centralized* engine or in a *decentralized* fashion. Decentralized orchestration improves scalability and concurrency. Dynamic binding coupled with decentralized orchestration adds high availability and fault tolerance to the system. In this paper, we categorize different forms of concurrency and provide an algorithm to identify these forms in a composite service specification. We also consider the impact of dynamic binding and faults on synchronization constructs.

Keywords

Web services, decentralized orchestration, synchronization

1. INTRODUCTION

Web services encapsulate information, software or other resources, and make them available over the network via standard interfaces and protocols. Complex web services may be created by aggregating the functionality provided by simpler ones. This is referred to as *service composition.* Typically a *composite* service is defined using an XML-based specification language such as BPEL[1] and its execution is driven by a centralized engine that interprets the workflow specification.

Runtime Architecture. Figure 1(a) shows the centralized orchestration of a *FindRoute* service, which sends a name, *name*1, to *AddrBook*(1) and a name, *name*2, to *AddrBook*-(2). The two addresses returned are sent to the *RoadRoute* service which returns the driving directions from one address to the other.

[1]http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

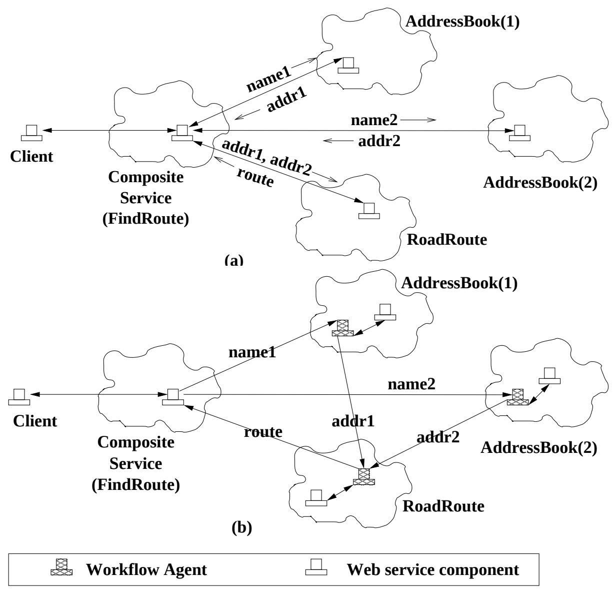

Figure 1: Centralized and Decentralized Architecture

In decentralized orchestration, rather than send every message through a single centralized server, messages are sent directly from the component where the data is generated to the component where the data is needed. For example, the address generated at *AddressBook*(1) is sent asynchronously to *RoadRoute*. A decentralized orchestration (Figure 1(b)) requires an engine or agent at or near the site of each component service. The agent has the following logic:

- Receive data/control messages from preceding components
- Execute business logic
- Invoke the component web service
- Evaluate exit conditions and asynchronously send data / control messages to succeeding component(s).

All messages between agents are asynchronous, whereas the web service invocation is synchronous. The composite service also behaves like an agent and sends and receives asynchronous requests from the component agents.

Dynamic Binding. Load distribution based on replicated services has become the standard way to improve the scalability and availability of Internet services. The services may be replicated within a local cluster [6] or the replicas may be geographically distributed [19]. The process of selecting one of the instances at runtime is termed dynamic binding. The actual instance invoked may be determined dynamically based on criteria such as request content [13, 2], scheduling algorithms [11] or geographical location [16].

Motivation. Given a composite service specification, there are two ways to improve its performance:
• parallelize components where the dependencies permit,
• decentralize the orchestration.

In Figure 1(b), there is clearly a synchronization overhead involving correlation of data from two different services at the `RoadRoute` service. However, this is offset by the reduced data flowing through the network and the additional concurrency. Experimental analysis shows that, in general, decentralized execution brings performance benefits, whereas with parallelization, the performance benefits may be offset by synchronization overheads. More details will be presented in Section 4. The advantages of decentralized execution are:
• The centralized engine does not become a performance bottleneck.
• The data flowing through the network is reduced as data flows only along data and control dependence edges.
• Improved concurrency brings improved performance.
• Asynchronous messaging results in better throughput and graceful degradation [7].

The advantages of dynamic binding are:
• It supports fault tolerance. If one service is not available, another equivalent service may be chosen at runtime.
• A service may be chosen on the basis of the price it charges or the response time it guarantees. This helps the composite service in conforming to its QoS commitments.
• It permits load balancing across the competing services and hence may improve performance.

1.1 Problem Description

We are building a tool that takes as input a centralized web service specification and converts it into a decentralized specification. Currently, the input is an XML specification which is a subset of BPEL.[2] The output is an XML specification (in the same language) for decentralized execution. We have also built the workflow agent which takes the XML specification as input and executes it. Converting a centralized specification to a decentralized specification requires three steps (i) Automatic Parallelization and Code Partitioning (ii) Synchronization analysis and (iii) Generation of the decentralized specification. We use a variation of Sarkar's algorithm for code partitioning [15] but the details are beyond the scope of this paper. The code partitioning determines *what* data needs to flow between components. The synchronization analysis determines *when* the data will be transferred. Both analyses are important as the output specification must make explicit both the *interaction* between components as well as the *synchronization mechanism.*

Synchronization analysis. Web services support message passing and RPC for component interaction. Based on these protocols, some of the standard models of synchronization are the *Postal* model and the *Deposit/Fetch* model [17]. Concurrency adds complexity to coordination. Hence, variations of the standard synchronization protocols have to be used. Different forms of concurrency based on the program structure need to be handled differently. We categorize the different forms of concurrency, and give an algorithm which uses *static analysis* to detect the different forms of concurrency in a given composite service. We also specify how synchronization is to be achieved in each case.

Dynamic Binding. Let there be several instances (with different access URLs) of the component service `RoadRoute` in the example in Figure 1(b). In the presence of dynamic binding, at runtime the two `AddressBook` services must choose one instance of `RoadRoute` to which they will send the addresses. Clearly, they must both choose the *same* instance, or the workflow will never complete. So an arbitration protocol is required between the two `AddressBook` services. Coordination for handling dynamic binding is also dependent on the type of concurrency.

Contributions of this paper. In this paper we
• identify coordination problems in decentralized execution of composite services,
• give solutions to the problem of coordination,
• experimentally evaluate the different solutions.

2. WORKFLOW DEPENDENCE GRAPHS

A *Workflow Dependence Graph* (WDG) is a graphical representation of a decentralized, concurrent orchestration. It can be obtained from a *Control Flow Graph* (CFG) [1] representing the workflow *specification* itself. The CFG of the input specification is a directed graph in which each node represents a task in the workflow specification. Each task may be an invocation of a workflow component web service or a part of the logic glue that binds the components into a composite whole. The CFG has a unique *Start* node that has no predecessors, and a unique *Exit* node with no successors. Every node is reachable from the *Start* node, and the *Exit* node is reachable from every node.
The *Program Dependence Graph* [12] (PDG) is a graph in which the nodes are the statements and predicate expressions of the CFG and the edges are data dependence and control dependence edges.
Data Dependence: A node N_j is data dependent on a node N_i, if N_i defines some data x, N_j uses the data x, and there exists a path from N_i to N_j without intervening definitions of x [1].
Postdominance: A node N_j *postdominates* a node N_i *iff* $N_i \neq N_j$ and N_j is on every path from N_i to *Exit*. A node N_j postdominates a branch of a predicate N_i *iff* N_j is the successor of N_i in that branch or N_j postdominates the successor of N_i in that branch.
Control Dependence: A node N_j is control dependent on a predicate N_i *iff* N_j postdominates a branch of N_i but N_j

[2]Although standard workflow specifications are not designed for decentralized orchestration, we found that BPEL, in particular, can be used for decentralized specification as well. In this paper we do not give details of the XML specification.

does not postdominate N_i.

We partition the PDG to generate a WDG. Whereas the PDG has one node for every task in the composite service, the WDG has one node for each component in the workflow and all the tasks in the PDG are partitioned across the WDG nodes so as to minimize the number of inter-component messages [10]. The edges of the WDG are inter-component data and control dependence edges. Data dependence edges indicate the data to be transferred from one component to another. The implication of a control dependence edge from N_i to N_j is that N_i *must* execute before N_j may execute.

Once the WDG has been computed, a decentralized orchestration can *follow WDG edges rather than control flow edges.*

3. SYNCHRONIZATION

We consider two basic synchronization protocols. In the *Direct Deposit* model, the sender deposits the data directly into the receiver's buffers. In the *Request-Response* model, data is buffered at the sender, a control message is sent indicating availability of the data, and the receiver pulls the data when required.

3.1 Concurrency

In the presence of concurrency, a node may have two or more incoming edges. In this section, we examine the effect of concurrency on synchronization. We identify the different forms of concurrency and describe the different ways to manage synchronization. We first analyze sequential code in the CFG and determine the forms of concurrency possible; then we analyze code with branches and finally code with iteration.

Sequential Code. For a WDG generated from a CFG with only sequential flow, a node in the WDG may have two incoming edges if it has two incoming data dependence edges as shown in Figure 2. Let N_3 be a node with two incoming data dependence edges $N_1 \rightarrow N_3$ and $N_2 \rightarrow N_3$. Then we classify the possible forms of concurrency as α concurrency and β concurrency on the following basis: in α concurrency there is **no** path from N_1 to N_2 in the WDG, whereas in β concurrency, there **exists** a path from N_1 to N_2 in the WDG. The corresponding synchronization protocols are visually depicted in Figure 2 and we briefly explain the protocols for β concurrency below.

- Direct Deposit: N_1 and N_2 directly and asynchronously send data to N_3.
- Request-Response: N_1 and N_2 send control messages to N_3; and N_3 pulls the data from N_1 and N_2 after receiving both the control messages.
- Combined Direct Deposit and Request-Response (CDDRR): N_2 does a direct deposit at N_3 and N_3 pulls data from N_1. This is possible since there is a path from N_1 to N_2 indicating that data is ready at N_1. Hence a control message from N_1 to N_3 is not required.
- Pipeline: This option dispenses with concurrency by pipelining the data from N_1 to N_3 through N_2.

The CDDRR and Pipeline options are not available in α concurrency since there is no path from N_1 to N_2 in the WDG.

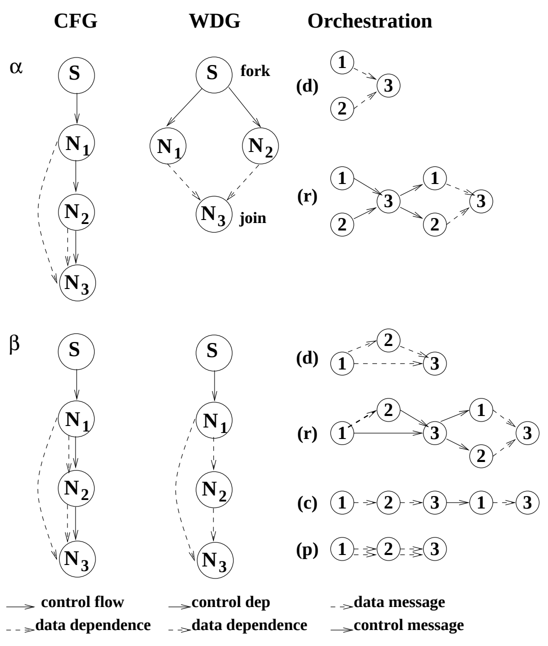

Figure 2: Synchronization protocols (d) for Direct Deposit, (r) for Request Response, (c) for Combined Direct Deposit and Request Response, and (p) for Pipeline, for WDGs generated from sequential code.

Branched Code. When there is a branch node in the CFG, the different forms of concurrency possible are o, α_1, β_1, α_2, and β_2 concurrency as shown in Figure 3.

In o concurrency, although there are two dependences reaching N_3, at run time only one path will be executed, and hence N_3 has only one dependence at runtime and there is no concurrency.

In α_1 and β_1 concurrency, we observe that N_1 does not know whether N_3 or N_3' will be executed and hence cannot do a direct deposit until N_2 has completed. This happens because N_3 does not post-dominate N_1 in the CFG. The modified protocol for Direct Deposit is shown in Figure 3. The other synchronization protocols do not differ from the corresponding α and β concurrency protocols.

In α_2 and β_2 concurrency, N_3 does post-dominate N_1, but the data generated at N_1 may get killed along some path to N_3. Hence in this case also N_1 cannot do a direct deposit at N_3. In addition, the data generated at N_1 may have to be discarded depending on the path taken at runtime. The protocols are visually depicted in Figure 3.

Iteration. A loop header may be treated in the same way as a branch node. In the absence of loop-carried dependences, iteration does not add any further complexity to the synchronization. Since a loop defines a strongly connected region wherein there is a path from every node to every other node in the region, in the case of loop-carried dependence, some form of β concurrency will apply.

3.2 Dynamic Binding

The different coordination schemes for dynamic binding are shown in Figure 4. The possible coordination schemes depend on the synchronization protocol used and the type of

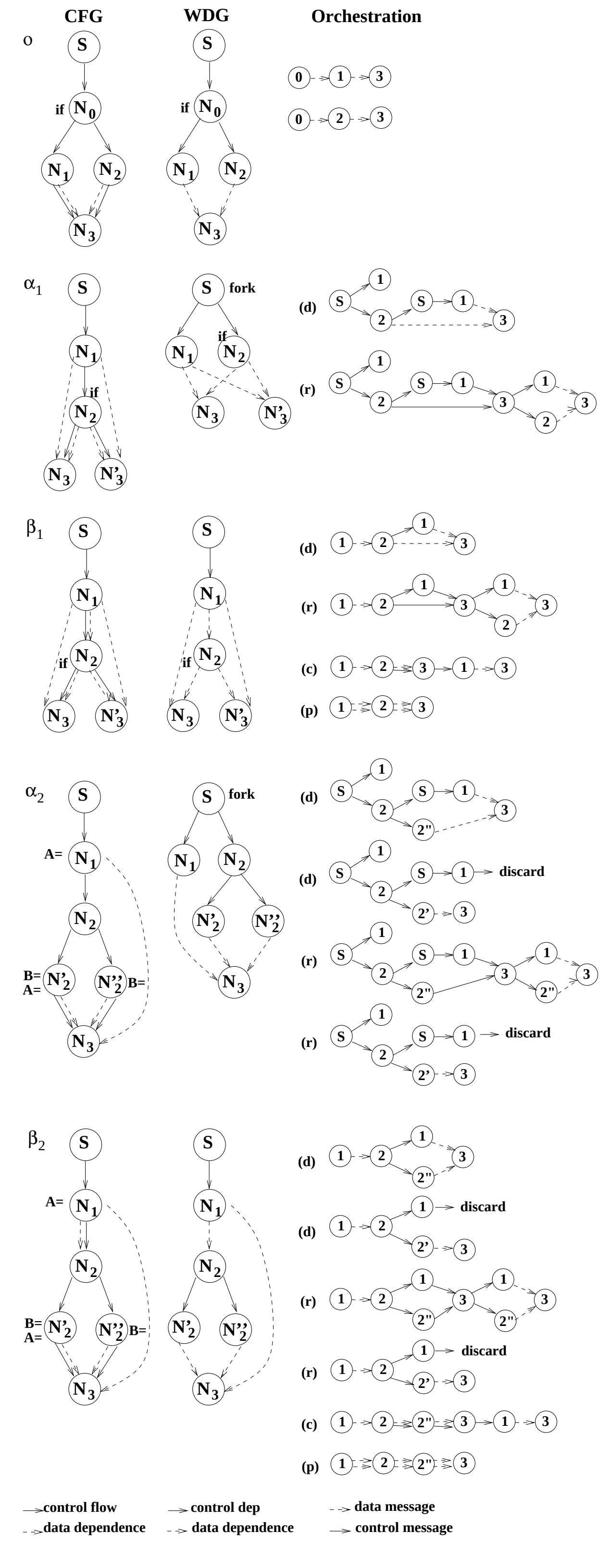

Figure 3: Synchronization protocols (d) for Direct Deposit, (r) for Request Response, (c) for Combined Direct Deposit and Request Response, and (p) for Pipeline, for WDGs generated from branched code.

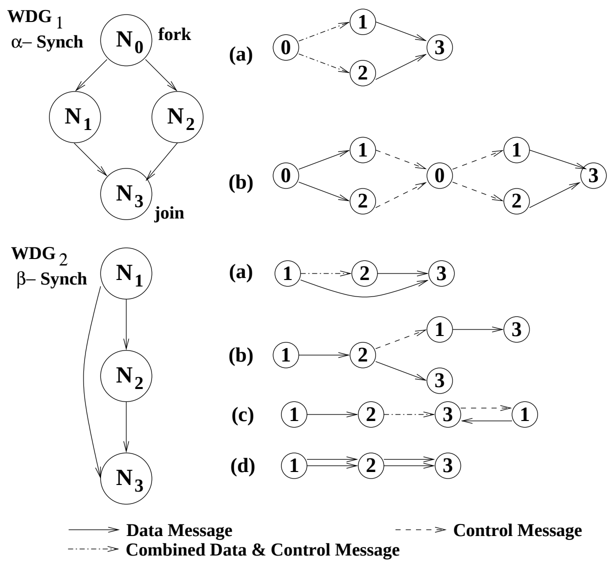

Figure 4: Coordination in Dynamic binding

concurrency. We now show how to handle dynamic binding for α and β concurrency. The other cases are similar.

α *Concurrency.* N_1 and N_2 need to arbitrate to decide which instance of N_3 to use. Since the actual instances of N_1 and N_2 themselves may have been decided dynamically, it is not possible for N_1 and N_2 to know each others addresses. However, N_0 is a common ancestor of N_1 and N_2 in the WDG, and hence the address of N_0 can be propagated down to both N_1 and N_2 and N_0 can be the arbitrator. There are two possible schemes of arbitration:
(a) N_0 makes the dynamic binding decision and passes the URL of the chosen instance of N_3 to N_1 and N_2 as control messages that may be combined with data messages. Then no further arbitration is required.
(b) When N_1 and N_2 are ready to send data to N_3, they make a dynamic binding decision to bind to (say) $Instance_1$ and $Instance_2$ respectively and inform N_0 using a control message. If N_0 receives N_1's request first then it accepts that decision and sends an accept to N_1 and when N_2's request arrives, it replies with the first decision taken so that N_2 knows it must use $Instance_1$ instead. Thus, N_1 and N_2 synchronize, using N_0 as an arbitrator.
These schemes apply to both Direct Deposit and Request Response synchronization. Option (b) may appear more complex than (a) but it has certain advantages as we will see in Section 3.3.

β *Concurrency.* WDG_2 in Figure 4 enumerates the different ways in which N_1 and N_2 may arbitrate. Option (a) and (b) are similar to (a) and (b) in α concurrency and can be used on top of Direct Deposit and Request Response. Option (c) is used in the case of Combined Direct Deposit and Request Response. N_2 takes the dynamic binding decision and N_3 pulls the data from N_1. Option (d) depicts the Pipeline case. There is no concurrency and hence no arbitration.

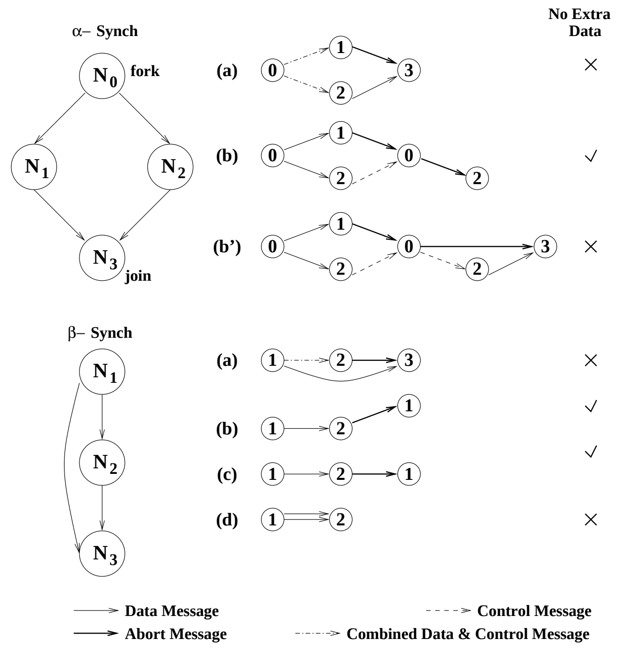

Figure 5: Error Management

```
Input: Nodes $N_1$, $N_2$, $N_3$ // $N_1$ and $N_2$ send data to $N_3$
Output: Type of concurrency
if $Path(N_1, N_2) ==$ false then
   $N_c = CommonAncestor(N_1, N_2)$
   if $N_c$ is a conditional node then
      $o$ concurrency
   else // $\alpha$ concurrency
      if $PostDom(N_3, N_1)$ and
         $!Killed(N_1, N_2, N_3)$ then
         $\alpha$ concurrency
      else if $!PostDom(N_3, N_1)$ and
         $!Killed(N_1, N_2, N_3)$ then
         $\alpha_1$ concurrency
      else
         $\alpha_2$ concurrency
else // $\beta$ concurrency
   if $PostDom(N_3, N_1)$ and
      $!Killed(N_1, N_2, N_3)$ then
      $\beta$ concurrency
   else if $!PostDom(N_3, N_1)$ and
      $!Killed(N_1, N_2, N_3)$ then
      $\beta_1$ concurrency
   else
      $\beta_2$ concurrency
```

Figure 6: Concurrency analysis

3.3 Distributed Error Management

There are two typical conditions under which the workflow may have to be terminated:

A synchronization problem: In situations where two nodes need to synchronize, if the join node fails after receiving one message there is a possibility that the workflow may have to abort.

Condition violation: A client may specify conditions under which a workflow must be aborted. For example, if the price of available movie tickets for "Lord of the Rings" exceeds $7, then abort the workflow.

Handling workflow abort. There are several options for a component when it detects a condition that calls for an abort - it may ignore the situation, it may send a message directly to the request initiator, or it may send a message back along the path on which the request arrived. Although exceptions are usually handled by propagating the error back along the call graph, in decentralized orchestration sometimes it makes sense to propagate the error *forward*. In Figure 5 we depict what happens when a node detects an abort condition.

α concurrency.

• In case (a) N_1 sends an abort message to N_3 but it is not possible to inform N_2 about the abort and hence the data flows from N_2 to N_3 where it is discarded.

• In case (b) there are two synchronization situations that may arise that have been depicted as (b) and (b') in Figure 5. In case (b), the abort message from N_1 to N_0 reaches before the synchronization request from N_2 to N_0. Hence, N_2 can be informed about the abort and it will send no data to N_3. However, (as depicted in case (b')) if the abort message from N_1 to N_0 reaches after the synchronization request from N_2 to N_0, then N_3 needs to be informed that it will receive no message from N_1. Therefore, N_0 needs to send an abort message to N_3 which can then discard any messages related to this request.

β concurrency.

• In case (a) N_2 needs to send a message to N_3 indicating that the request has been aborted. Else, N_3 will receive data from N_1 and keep waiting for the second data message from N_2 (presumably until some predetermined timeout), thus wasting resources.

• In case (b) and (c) N_2 sends a message back along the call graph to N_1 since N_1 is holding data to be sent to N_3. N_1 can now discard the data and send a message directly to the request initiator indicating an abort.

Thus we see that in case (a) it is important to propagate the error message *forward* along the call graph, rather than backwards. Note however, that forward propagation of errors does not go through every succeeding node in the graph but only through the nesting "join" nodes. For β concurrency, in terms of efficiency, we note that in case (b) and case (c) no unnecessary data is put on the network, whereas in case (a), the data is sent from N_1 to N_3 and then discarded. In case (d), unnecessary data flows from N_1 to N_2 where it is discarded. Thus the more complex coordination protocol fares better in the presence of faults.

3.4 The Algorithm

Since our source program is a structured program (without unconditional `goto`s and without irreducible loops), all forms of concurrency can be treated as a combination of these basic constructs. If there are more than two dependence edges at a node they can be evaluated pair-wise and each pair will fall into one of the above categories.

The algorithm for determining the type of concurrency is given in Figure 6. N_3 is the `join` node with two incoming edges from N_1 and N_2. For each pair of incoming edges the algorithm in Figure 6 is executed.

The function $CommonAncestor(N_i, N_j)$ returns the closest common ancestor of N_i and N_j in the WDG. The function $PostDom(N_i, N_j)$ returns true if N_i postdominates N_j in the CFG and false otherwise. The function $Killed(N_i, N_j, N_k)$ returns true if the definition at N_i is killed along the path $N_i \rightarrow N_j \rightarrow N_k$ in the CFG. Note that $CommonAncestor$ and $Path$ are computed on the WDG but $PostDom$ and $Killed$ are computed on the original CFG.

Having determined the type of concurrency in the composite service, we can then use the appropriate decentralized orchestration as shown in Figures 2–5.

4. EXPERIMENTAL RESULTS

4.1 Experimental setup

We have implemented a workflow agent for decentralized composite service execution. Each node has one such agent, capable of receiving and sending *asynchronous* messages over HTTP. At runtime the agent interprets a decentralized specification coded in a subset of BPEL.

Requests are initiated by clients that send a *synchronous* HTTP message to a composite service. The composite service holds the client's connection and *asynchronously* dispatches data to the relevant component service(s). The last component asynchronously returns a response to the composite service, which forwards the response to the client over its pending connection. We vary the load on the system by varying the number of concurrent clients.

We conducted experiments both in a LAN environment as well as on a WAN. All the web services run on standard Intel-based machines running `tomcat` on Linux. In the LAN setting, the machines are networked using 100Mbps Ethernet. For the WAN experiments, we used machines distributed between locations in India, New York and California.

4.2 Results and Analysis

We ran the experiments with dummy "echo" services that merely echo back the request. To experiment with different message sizes, we gave the echo service a second parameter indicating the required length of the output message. We experimented with message sizes ranging from about 2000 bytes to 20000 bytes. The size of HTTP messages is typically in the range of 1K to 10K bytes but we feel that web services may have larger message sizes especially due to XML-izing of the data. We have plotted the average response time for each of the coordination models as a function of the message length. The plots for both the LAN-based and WAN-based experiments are shown in Figure 7. We discuss the plots below.

• In the LAN environment we found that the response time is very sensitive to the size of the message.
- α concurrency: Direct Deposit is a clear winner. The centralized orchestration is the worst performer at lower message lengths, but at higher message lengths it gives better results than the Request-Response option. The Request-Response option does not scale well with increasing message length due to the large number of connections and lack of data concurrency
- β concurrency: At low loads the Pipeline option and Combined Direct Deposit and Request-Response options actually perform better than the Direct Deposit option due to its low synchronization requirements. But as the load increases we observe that Direct Deposit scales the best due to its high support for concurrency. Results for β_1 concurrency are very similar to β concurrency
- β_2 concurrency: The modified Direct Deposit option has significant coordination overheads. The Pipeline actually scales and performs better due to the lower synchronization and connection requirements, despite the overhead of extra data.

• In the WAN environment the increase of response time with message length was not so marked.
- α concurrency: Direct Deposit is a clear winner but the Request-response option also easily outperforms the Centralized orchestration. The network transmission is the single largest overhead in the system. The centralized orchestration does not scale well.
- β concurrency: The Pipeline option with minimum synchronization overheads leads despite carrying extra data. All the decentralized options scale well. Only the centralized option does not scale well.
- β_2 concurrency: In this case the Pipeline's performance drops drastically since the pipeline is across three connections ($N_1 \rightarrow N_2 \rightarrow N_2'' \rightarrow N_3$) all of which were inter-continent links. In the β concurrency pipeline only one of the links was an inter-continent link. This shows that the pipeline is useful only if the distances, data and number of links are small.

In general, in the WAN environment, the network transmission is the largest overhead. In the LAN environment network delays are small, and the number of connections governs the scalability of an orchestration. In both settings, decentralization clearly brings performance benefits. Pipelines are useful only over small distances and for small messages. Otherwise, concurrency brings greater performance.

Dynamic Binding and Error Management: Preliminary experimental evaluation shows that the arbitration option (a) gives the best performance especially as it is typically coupled with the efficient Direct Deposit synchronization. We need to determine the effect of faults on this policy as it likely to perform poorly in the presence of faults. However, if we assume that faults are rare (which is not unrealistic), then this scheme should continue to give the best performance.

5. RELATED WORK

Much work has been done in analysis of synchronization models and in coordination. Stricker *et al.* [17] analyze the execution costs of different synchronization protocols, Mehta *et al* [9] have come up with a complete taxonomy of software connectors. Amongst others, they identify the different styles applicable to middleware and distributed systems. Ciancarini *et al.* [5] focus on the role of XML in coordinating middleware. XML plays a strong role in our tool also, although this paper does not go into the details. Cabri *et al* [4] focus on coordination for mobile agents.

Distributed workflows have been around for some time. Tripathi *et al.* [18] focus on XML specification that is interpreted by mobile agents. Kim *et al.* [8] have modeling methodology to analyze the scalability and performance of centralized versus distributed data and/or control dependence. Ponnekanti *et al.* [14] have an interesting rule-based algorithm that permits semi-automation of workflow composition. Atluri *et al.* [3] discuss security issues in distributed

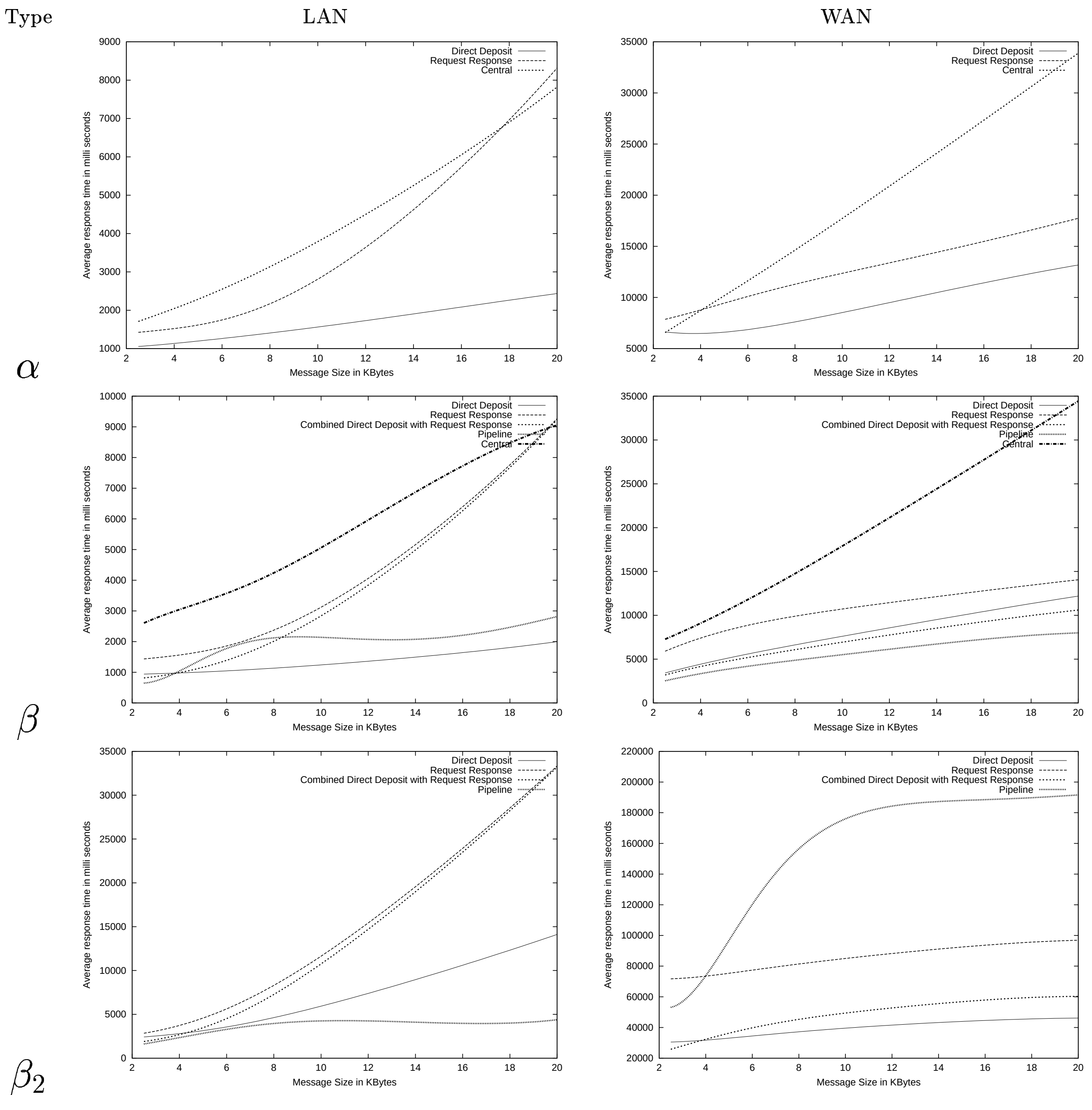

Figure 7: Experimental results

workflows. The *Workflow Management Coalition* (WfMC) has proposed a reference architecture and defined interfaces for vendors. While this is a useful step toward interoperability, the current specification is oriented towards centralized engines and does not provide support for decentralized engines with dynamic binding. To the best of our knowledge, no one else has addressed the issues of dynamic binding and fault handling in decentralized workflows, or the resultant synchronization problems.

6. SUMMARY AND CONCLUSION

In general, decentralized systems have more complex coordination requirements than other distributed systems. In this paper we have focused on issues related to synchronization and coordination between components of a decentralized composite web service. We first determined the different types of concurrency that may be found in a decentralized setting. Then we determined the different possible ways to coordinate the components. Finally we experimentally evaluated each of the options. In general, our experiments demonstrate that decentralization improves performance and scalability.

Future Work We are currently working on building a dynamically configurable system that reconfigures itself based on monitoring feedback. The feedback can be used to determine the best selection policy for dynamic binding, and to determine the appropriate synchronization protocol for each concurrency point in the system.

Limitations Decentralized orchestration works on the assumption that there is an agent at or near each distributed node, which is capable of executing the generated XML specification.

7. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based netwrok servers. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

[3] V. Atluri, S. A. Chun, and P. Mazzoleni. A chinese wall security model for decentralized workflow systems. In *Proc. of the Conference on Computer and Communications Security*, November 2001.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. Coordination models for internet applications based on mobile agents. *IEEE Computer Magazine*, 1999.

[5] P. Ciancarini, R. Tolksdorf, and F. Zambonelli. Coordination middleware for xml-centric applications. In *Proceedings of the ACM Symposium on Applied Computing*, 2002.

[6] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the ACM Symposium of Operating Systems (SOSP)*, October 1997.

[7] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the Symposium on Operating Systems Design and Implementation(OSDI2000)*, October 2000.

[8] K.-H. Kim and C. A. Ellis. Workflow performance and scalability analysis using the layered queuing modeling methodology. In *Proceedings of the International Conference on Supporting Group Work*, 2001.

[9] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering(ICSE 2000)*, May 2000.

[10] M. G. Nanda and S. Chandra. Decentralizing composite web services. In preparation.

[11] J. Nieh, C. Vaill, and H. Zhong. Virtual-time round-robin: An O(1) proportional share scheduler. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[12] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACMSIGSOFT, 1984.

[13] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, October 1998.

[14] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for building composite web services. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[15] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35:779–804, 1991.

[16] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. In *Proceedings of the ACM Workshop on Internet Server Performance (WISP'98)*, 1998.

[17] T. Stricker, J. Stichnoth, D. O. Hallaron, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proceedings of the ACM International Conference on Supercomputing*, 1995.

[18] A. Tripathi, T. Ahmed, V. Kakani, and S. Jaman. Distributed Collaborations using Network Mobile Agents. In *2nd International Symposium on Agent Systems and Applcations/4th International Symposium on Mobile Agents*, September 2000.

[19] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proceedings of the ACM Symposium of Operating Systems*, 2001.