Optimal Call based Checkpointing for Orchestrated Web Services

A.Vani Vathsala
Department of Computer Science
CVR college of Engineering and University of Hyderabad

ABSTRACT
Web Services are built on service-oriented architecture which is based on the notion of building applications by discovering and orchestrating services available on the web. Complex business processes can be realized by discovering and orchestrating already available services on the web. In order to make these orchestrated web services resilient to faults, we proposed a simple and elegant checkpointing policy called "Call based Global Checkpointing of Orchestrated web services" which specifies that when a web service calls another web service the calling web service has to save its state. But performance of the web services implementing this policy reduces due to checkpointing overhead. In an effort to improve this policy, we propose in this paper, a checkpointing policy which uses Predicted Execution Time and Mean Time Between Failures of the called web services to make checkpointing decisions. This policy aims at reducing the required number of Call based Checkpoints but at the same time maintains the resilience of web services to faults.

General Terms
Web Services

Keywords
Checkpoints, Web Services, Mean Time Between Failures, Orchestration.

1. INTRODUCTION
A service in execution may take service of another service and this may result in nested call of services. This is known as Orchestration of services. In case of such service execution pattern, if a service fails to complete (for any possible reason) then all the services dependent on the failed service are to be re-executed causing a voluminous rework.

Traditionally such a situation is handled (for avoiding rework) by checkpointing. In our earlier work, [5], we have proposed “Call Based Checkpointing Policy” that saves status of caller services so that in case of failure of callee service, the computation at the former can be resumed at this saved point. However this method is time consuming due to the overhead of maintaining caller status at every service call.

In this work, we propose a method that does not necessitate checkpointing at every call thus reducing instances of checkpointing. The rationality on decision making is based on two factors i.e., Execution Time Prediction and Mean Time Between Failures. A caller service predicts the execution time of the callee S₂, say PET(S₂). This is a possible estimate from the execution history of callee services. Let the Mean Time Between Failures of the callee be MTBF(S₂). If PET(S₂)<MTBF(S₂) then the caller most probably can avail the service from the callee. Hence checkpointing the caller at the service call is not required. This paper details on this concept and advocates its utility for orchestrated services in making them resilient to possible errors.

If S₁ is a composite web service, a call to S₁ might result in a nested call; whether to take a checkpoint or not while calling each of the services involved in the nested call, has to be decided. This decision at each step has to be taken without needing many computations (Execution Time predictions of the services involved in the nested call). Hence we have proposed to use the already available computations, i.e., PET of S₁ and composition operators used to compose involved web services, to take the decisions. We have proved the fact that PET of the composite service S₁ and knowledge of composition operators alone are sufficient to take these decisions.

In section 2 we present our analysis of work done in this area. In section 3 we present our basic Call based Checkpointing policy and in section 4 we give a detailed description of our Execution Prediction based checkpointing policy. In subsection 4.1 of this section we give briefly the method for calculating MTBF for web services. In subsection 4.2 we describe the method of using Euclidean distances to predict Execution time of web services. In subsection 4.3 we describe how to minimize the number of checkpoints to be taken. We also discuss the role of composition operators and PET of a composite service in making Call based Checkpointing decisions with necessary proofs. Towards the end of this section we have demonstrated the generation of Global checkpoints of an orchestrated web service using the new Execution Prediction Based Checkpointing policy. We conclude by giving a sketch of our future work.

2. RELATED WORK
Few papers [6,7,8] have been published discussing the need and techniques for checkpointing web services. But all these works require the user to specify the exact checkpointing locations. In contrast we proposed a simple and elegant checkpointing policy[5] for orchestrated web services which specifies that whenever a web service calls another web service, the state of calling web service must be saved. But checkpointing web services at all pre specified locations (at all service calls) may lead to overzealous checkpointing that results in degradation of the performance altogether. Hence to improve the performance of composed web services with call based checkpoints, we propose Execution Prediction based Checkpointing scheme.

The research works presented in [2,3,4] propose methods for predicting runtime of web services. These works advocate the use of predicted execution time for selection of web services to construct composite web service workflows. To the best of our knowledge there is no work which concentrates on using
Zoltan Balogh et.al presented a knowledge based approach\[2\] for predicting runtime of stateful web services. To predict the execution time of a web service instance, it maintains a knowledge base of possible different past cases for different combinations of input parameters. Given a web service instance, Euclidean distances are used to find out most similar past cases. The runtime for the given web service instance is predicted to be the average output value of the most similar past cases. Estimation of web services runtime is done keeping in view construction of composite web service workflows.

Zhengdong Gao uses Back Propagation Neural Networks to predict the runtime of a given web service\[4\]. He uses Availability, Network Bandwidth, Response Time, Reliability of the given web service as inputs to the Neural Network which produces predicted execution duration as output. The core of his work is the design and implementation of BP Neural Network which is used to predict performance of services.

In order to predict timing failures, Laranjeiro \[3\] proposes to use a graph based approach. He analyzes the service code and builds a graph to represent its logical structure. He then gathers time-related performance metrics during runtime. This data is used to predict if a given execution will or will not conclude in due time.

The research presented in \[1\] intends to provide the concept of MTTF(Mean Time To Failure) of composite web service. It describes the calculation method of MTTF of composite web services based on the workflow composition pattern. The authors use the concept of MTTF of web services to find out reliability of a given composite web service.

3. CALL BASED CHECKPOINTING POLICY

Calling a web service includes the following steps:

At Service requestor side:
1) Initiation of the call.
2) Encoding data to be transmitted as a SOAP message
3) Transmitting the SOAP message

At Service provider side:
1) Receiving the SOAP message
2) Decoding the SOAP message
3) Parsing the decoded XML data to locate the actual method name and the parameters.

Thus we can see that calling a web service includes several steps and incurs considerable cost and time at run time. When the calls are nested and if there is any kind of failure the entire sequence of calls has to be re-invoked causing considerably delay in response which results in degradation of quality of the service provided.

Motivating example
A customer requests a web service for his loan processing. This loan processing web service receives the request from the customer which consists of information like his name and requested credit amount. This loan processing web service, \(S_0\), invokes two more web services: Loan approver web service \(S_1\) and accessor web service \(S_2\). If the amount requested is less than 10,000 \(S_0\) calls the loan accessor web service \(S_2\). This web service, based on some customer records, decides and reports back whether there is high risk in approving loan to the customer. If \(S_2\) reports low risk, \(S_0\) itself approves the loan. If the amount requested is greater than 10,000 or if \(S_2\) reports high risk, the loan approver web service \(S_1\), is invoked by \(S_0\) to enquire about the customer and report whether to approve the loan to the customer or not. If the amount requested is greater than 1,00,000 \(S_1\) outsources the job to another web service \(S_3\). \(S_1\) calls another loan approver web service \(S_4\) to take second opinion and sends back the reply to \(S_0\). Finally loan processing web service \(S_0\) sends back its reply to the customer. Fig 1 depicts an execution instance of this loan processing web service.

In Fig 1, if \(S_4\) fails at time \(t_1\), then all the invocations have to be repeated, if the application does not use any checkpointing policy.

Fig 1: An Execution Instance

Our checkpointing policy aims at avoiding expensive re-invocations of web services and hence we propose that Checkpoints must be taken when web services interact with each other. Web services interact with each other by invoking any of the operations specified in corresponding WSDL files. A WSDL file defines four types of operations:

One-way: A web service receives a message but does not return a response.

Request-response: A web service receives a request and returns a response.

Solicit-response: A web service sends a request and waits for a response.

Notification: A web service sends a message but does not wait for a response.

We propose a call based checkpointing policy which is presented in Table 1.
Table 1: Checkpointing Policy

<table>
<thead>
<tr>
<th>Type of Operation</th>
<th>Checkpointing Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request-Response / Solicit-Response</td>
<td>Save the state of the service requestor after sending the request. Log the message at service provider side upon receiving the request. Similarly save the state of the service provider after sending the response and log the response message after it is received by the service requestor</td>
</tr>
<tr>
<td>One Way / Notification</td>
<td>Save the state of the sender after sending the message and log the message at receiver side after receiving it.</td>
</tr>
</tbody>
</table>

When a synchronous communication is initiated between a service requestor and a service provider, the type of operation becomes ‘Solicit-response’ at the service requestor side and it becomes ‘Request-response’ at the service provider side. Therefore, the checkpointing policy is same for these two operations. Fig 2 demonstrates our checkpointing policy for synchronous interactions. When S0 invokes S1 and is waiting for its reply, S0 blocks itself and cannot do any useful work.

Hence saving the state of service requestor upon receiving the reply from service provider is not required and it is sufficient to log the reply message for later replay.

Fig 2: Checkpointing policy for Synchronous interactions

When an asynchronous message is sent by a sender to a receiver, the type of operation becomes ‘Notification’ at the sender side and it becomes ‘One Way’ operation at the receiver side. Hence we see that the checkpointing policy is same for these two operations also.

3.1 Call-based global checkpoints

A state of a web service in execution is characterized by the state of its local memory and a history of its activity. If such a state of the web service is saved on a stable storage, then the saved state is called as a local checkpoint for the web service. A local checkpoint that is taken most recently is called as the latest local checkpoint for the web service.

An orchestrated web service S0 is a composition of one or more constituent web services. It has pieces of code that it executes on itself and also calls other web services based on some conditions. If the orchestrated web service is not having any active calls (time t2 in Fig 1), and is executing its own piece of code, then its latest local checkpoint gives the latest global checkpoint of the composed application.

When S0 calls another web service, there may be a nested chain of service calls because of which more than one web service can be active(not completed their execution) at a given point of time.(at time t1 in Fig 1 web services S0, S1 and S2 are active). Hence the state of the orchestrated web service is collectively represented by the states of all active web services when a service call is in place. Thus, Call-based Global checkpoint for an orchestrated web service which has a service call in process, is defined as the set of the latest local checkpoints of each of the web services that are active during the call.

Let C0 represent the local checkpoints generated by S0. Let C1 represent local checkpoints generated by the service S1. C2 represent local checkpoints generated by the service S2 and so on.

S0 might make several web service calls, while in execution. Let C0im represents the checkpoint generated by S0 for mth service call that it has placed. Let S0 invoke S1 in its ith service call. Then C1jm represents the jth local checkpoint taken by S1 when it is serving ith call of S0.

To provide the service, if S1 makes use of services provided by other web services we have three superscripts in checkpoint numbering. In general, checkpoint Cim jm indicates: S0 invokes S1 as part of its ith service call, S1 invokes Sm as part of its jth service call, and this is the kth local checkpoint taken by web service Sm.

By applying the Call-based checkpointing policy to the execution instance depicted in Fig 1, we can see that number of local checkpoints generated as part of Call based Global checkpoint is ‘9’ as shown in Table 2.

Thus we see from Table 2 that when the service reply is received back by S0, the Call-based Global checkpoint reduces to the latest local checkpoint of S0.

Upon failure the application has to be rolled back to latest global checkpoint and all the messages received after the latest global checkpoint have to be replayed from the message logs. Execution of the composed application can thus continue from latest global checkpoint without re-invocation of already finished constituent web services.

Table 2: Local Checkpoints generated as part of Call-based Global Checkpoints (CBGC)

<table>
<thead>
<tr>
<th>Status of execution</th>
<th>CBGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke S2</td>
<td>{C0} /* first CBGC */</td>
</tr>
<tr>
<td>End S2</td>
<td>{C0, C1, C2} /*End of first CBGC */</td>
</tr>
</tbody>
</table>
A web service may be composed of a set of web services using the three primitive operations sequence, parallel and choice as depicted in Table 3. Fig 3 shows different cases of MTBF calculation. Other composition operations can be derived from these three primitive operations.

### Table 3: MTBF Calculation

<table>
<thead>
<tr>
<th>Composition Operation</th>
<th>MTBF Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence: S = S₁</td>
<td>Θ = λ₁ + λ₂ ; Θ = 1/λ</td>
</tr>
<tr>
<td>Parallel: S = S₁</td>
<td>Θ = λ₁ + λ₂ ; Θ = 1/λ</td>
</tr>
<tr>
<td>Choice: S = (S₁ + S₂)</td>
<td>λ₁P₁ + λ₂P₂ ; Θ = 1/λ ; S₁ is selected with probability P₁ and S₂ is selected with probability P₂</td>
</tr>
</tbody>
</table>

**4. EXECUTION PREDICTION BASED CHECKPOINTING OF WEB SERVICES**

In order to improve the performance of composed web services with call based checkpoints, we propose Execution Prediction based Checkpointing scheme. For each service call, this scheme decides, considering the PET and MTBF of the called web service, whether a checkpoint has to be taken on making a call to the service. Hence to implement this scheme, a caller should know the MTBF and PET of the callee.

MTBF of a service is an average measure of the time duration for which the service can run without failure. MTBF of a service has to be made public by the service itself by placing the MTBF in its WSDL. This MTBF can then be used by the service requestors to implement the checkpointing policy. When S₀ calls another web service S₁, PET of S₁ is calculated by S₀ using Euclidean distances method as explained in subsection '4.2' below.

**Checkpointing Rule:**

If PET (S₁) < MTBF (S₁), then S₁ will execute within its MTBF and eventually send back the reply to S₀. In such a case S₀ need not take a checkpoint while calling S₁ with anticipation that S₁ might fail.

Else if PET (S₁) ≥ MTBF (S₁) then S₁ might fail before sending a reply back to S₀ and hence S₀ must take a checkpoint before calling S₁.

**4.1 Calculation of MTBF**

Let λ represent the Failure rate of a web service S and let Θ represent the MTBF. Then Θ = 1/λ.

MTBF of a web service can be obtained by taking inverse of its Failure rate. Failure rate of a web service can be obtained by measuring its number of failures per unit time. (Ex: Failure rate = 5 failures in one hour. MTBF = 1/Failure rate = one hour/5 = 12 minutes).

**MTBF of composite services:** If a web service is a composition of other web services, then their MTBFs will affect the MTBF of the composite service. Let Θ represent the MTBF of a constituent web service Sᵢ where 1 ≤ i ≤ n and n is the maximum number of web services involved in the composition.

<table>
<thead>
<tr>
<th>Operation</th>
<th>MTBF Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke S₁</td>
<td>[Cᵢᵣ] /* second CBGC */</td>
</tr>
<tr>
<td>invoke S₃</td>
<td>[Cᵢᵢ, Cᵢ₋₁]</td>
</tr>
<tr>
<td>End S₃</td>
<td>[Cᵢᵢ, Cᵢ₋₁, Cᵢ₋₂]</td>
</tr>
<tr>
<td>invoke S₄</td>
<td>[Cᵢᵣ, Cᵢ₋₁]</td>
</tr>
<tr>
<td>End S₄</td>
<td>[Cᵢᵣ, Cᵢ₋₁, Cᵢ₋₂]</td>
</tr>
<tr>
<td>End S₁</td>
<td>[Cᵢᵣ] /* End of second CBGC */</td>
</tr>
<tr>
<td>End S₀</td>
<td>[Cᵢᵣ] /* third CBGC */</td>
</tr>
<tr>
<td>Total No of Local checkpoints generated = 9</td>
<td></td>
</tr>
</tbody>
</table>

**4.2 Execution Time Prediction**

Execution time Prediction can be done by comparing the current execution instance with similar previous cases. Input parameter values can be compared to find out the similarity between any two execution instances. Let I = {i₁, i₂, …, iₙ} represent an execution instance of a web service with m input parameters.

Euclidean distance can be used to find out the similarity. The similarity between any two cases I₁, I₂ is computed using the following formula for finding Euclidean Distance (ED).

ED (I₁, I₂) = SQRT (∑ᵢ₌₁ⁿ [i₁ᵢ - i₂ᵢ]²)

The case/instance which has the smallest Euclidean distance to the current execution instance is considered to be the most similar case.

Table 4 depicts history of execution instances of our loan processing web service, where m = 2, I = {LoanAmount, Risk}. These values are synthesized values and are based on the number of web services that will be invoked for the corresponding execution instance. Let the input parameter values for current execution instance of S₀ be Loan Amount = 1,270,000 and Risk = IR.

**Table 4: Execution instances of Loan Processing Example**

<table>
<thead>
<tr>
<th>Loan Amount</th>
<th>Risk</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>9000</td>
<td>Low</td>
<td>2 tu</td>
</tr>
<tr>
<td>8000</td>
<td>High</td>
<td>6 tu</td>
</tr>
<tr>
<td>25,000</td>
<td>IR</td>
<td>4 tu</td>
</tr>
<tr>
<td>1,25,000</td>
<td>IR</td>
<td>7 tu</td>
</tr>
<tr>
<td>7000</td>
<td>Low</td>
<td>2 tu</td>
</tr>
</tbody>
</table>

IR=Irrelevant I=Invoked tu=time units

While calculating Euclidean distances, input parameters having non-numerical values pose a problem. In such a case, map Non-numerical values to numerical values. For example, Input parameter 'Risk' in the example has fixed non-numeric values 'Irrelevant', 'Low', 'High'. They can be mapped to corresponding numeric values 1,2,3.
Similarly input parameters that have range values, like ‘LoanAmount’ in our example, where the range in which they fall is more important than the actual value, we have to map each range to a numerical value.

Calculation of Euclidean distances for this example reveals that there is one case similar to current execution instance. Take the average of execution times of similar cases to predict the execution time of the current execution instance, which is 7 time units.

4.3 Minimizing the number of Checkpoints

Goal of Execution prediction based Checkpointing policy is to minimize the number of local checkpoints that are generated as part of the Call-based Global checkpoint.

When a web service $S_0$ calls another web service $S_1$ and if $\text{PET} (S_0) < \text{MTBF} (S_1)$ then it indicates that the called web service completes its execution within its MTBF. Hence the calling web service will get its reply and there is no need to take a checkpoint in the calling web service. If the called service is a composite service and results in nested calls, decision has to be taken whether the checkpoints have been taken throughout the path of the nested call or not.

For example, if a composite service $S_2$ calls two constituent services $S_1$ and $S_3$ then it has to be decided whether $S_2$ must take checkpoints while calling $S_1$ and $S_3$. According to the policy, when $S_1$ calls $S_2$, $S_1$ must calculate $\text{PET} (S_2)$ and obtain MTBF of $S_2$ from WSDL file of $S_2$. If $\text{PET} (S_2) < \text{MTBF} (S_2)$ then there is no need to checkpoint $S_2$ while calling $S_3$. But in order to take this decision, $S_1$ must calculate the $\text{PET} (S_3)$. Similar is the case of calling $S_3$. These calculations can be avoided if the decision can be taken by using $\text{PET} (S_1)$ and $\text{MTBF} (S_1)$ alone.

We have that $\text{PET} (S_1) < \text{MTBF} (S_1)$. If it can be proved that $\text{PET} (S_2) < \text{PET} (S_1)$ and $\text{MTBF} (S_1) < \text{MTBF} (S_2)$ then it can be deduced that:

$$\text{PET} (S_2) < \text{PET} (S_1) < \text{MTBF} (S_1) < \text{MTBF} (S_2)$$

Hence $\text{PET} (S_2) < \text{MTBF} (S_2)$

Whether $\text{PET} (S_2)$ is less than $\text{MTBF} (S_2)$ or not can thus be found without calculating $\text{PET} (S_2)$.

\textbf{Proof:}

$\text{PET}$ of a composite service = It’s Local Computation time + Time taken to place Service calls + $\text{PET}$ of constituent services.

$\text{PET} (S_1) = \text{Local Computation time of} \ (S_1) + \text{Time taken to call} \ S_2 + \ S_1 + \text{PET} (S_2) + \text{PET} (S_1)$

\textit{Therefore} $\text{PET} (S_1) > \text{PET} (S_2)$ and $\text{PET} (S_1) > \text{PET} (S_3)$.

\textit{OR} $\text{PET} (S_2) < \text{PET} (S_1)$ and $\text{PET} (S_1) < \text{PET} (S_3)$.

Hence first half of equation (1) is proved. We have to consider MTBF calculations to prove second half.

MTBF for a composite service is calculated taking into account MTBF of its constituent services also, as shown in Table 3. Fig 3 explains MTBF calculation.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3}
\caption{MTBF Calculation}
\end{figure}
b) If \( S_1 \) is a composite service: policy is explained using the Fig 4 below for basic cases of composition and using Fig 5 for combination of composition operations.

For any combination of sequence and parallel composition operations, equation (1) holds good because in any case, \( \lambda_1 \) is the sum of \( \lambda \)'s of constituent services. The same is explained using Fig 5. But in case of choice composition operation, equation (1) does not hold good. Hence any sub composition involving 'choice' operation leaves us with no choice other than taking checkpoints for that sub composition.

But when Execution prediction based checkpointing policy is used, the number of checkpoints generated as part of Call based Global checkpoint reduces to 3 as shown in Table 5.

This is because when \( S_n \) calls \( S_1 \), \( S_1 \) calls \( S_2 \) and \( S_3 \) in sequence resulting in a nested call. Before \( S_n \) calls \( S_1 \), \( S_0 \) calculates that PET (\( S_1 \)) = 4tu and MTBF (\( S_1 \)) = 5tu. Hence it is found that PET (\( S_1 \)) < MTBF (\( S_1 \)) and \( S_0 \) need not take a checkpoint before calling \( S_1 \). \( S_1 \) calls \( S_3 \) and \( S_4 \) in sequence and since composition operation is 'sequence', we can see that equation (1) holds good and there is no need to calculate PET (\( S_3 \)) and PET (\( S_4 \)). Hence it can be deduced that PET (\( S_3 \)) < MTBF (\( S_3 \)) and PET (\( S_4 \)) < MTBF (\( S_4 \)) and \( S_1 \) need not take checkpoints while calling them. Also \( S_3 \) and \( S_4 \) need not take checkpoints after sending reply back to \( S_1 \) since their predicted execution time is less than their MTBF and they will not fail.

Thus we can see that total number of Local checkpoints generated is greatly reduced by implementing Execution time Prediction based Checkpointing.
Table 5: Demonstration of Execution Prediction based Checkpointing

<table>
<thead>
<tr>
<th>Current node visited</th>
<th>PET and MTBF Calculation by the caller</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke $S_2$</td>
<td>$\text{PET} (S_2) = 2tu$ and $\text{MTBF} (S_2) = 1tu.$</td>
<td>${C_0}$ generated by $S_0$. /* first CBGC */</td>
</tr>
<tr>
<td>End $S_2$</td>
<td></td>
<td>${C_0}$, $C_2^{11}/$ End of first CBGC */</td>
</tr>
<tr>
<td>invoke $S_1$</td>
<td>$\text{PET} (S_1) = 4tu$ and $\text{MTBF} (S_1) = 5tu.$</td>
<td>${C_0}$ not generated by $S_0$.</td>
</tr>
<tr>
<td>End $S_1$</td>
<td></td>
<td>$C_1^{21}$ not generated by $S_1$.</td>
</tr>
<tr>
<td>invoke $S_3$</td>
<td>$\text{PET} (S_3) &lt; \text{MTBF} (S_3)$ can be deduced from $\text{PET} (S_3) &lt; \text{MTBF} (S_3)$</td>
<td>$C_1^{21}$ not generated by $S_1$.</td>
</tr>
<tr>
<td>End $S_3$</td>
<td></td>
<td>$C_2^{21}$ not generated by $S_2$.</td>
</tr>
<tr>
<td>invoke $S_4$</td>
<td>$\text{PET} (S_4) &lt; \text{MTBF} (S_4)$ can be deduced from $\text{PET} (S_4) &lt; \text{MTBF} (S_4)$</td>
<td>$C_1^{22}$ not generated by $S_1$.</td>
</tr>
<tr>
<td>End $S_4$</td>
<td></td>
<td>$C_2^{21}$ not generated by $S_2$.</td>
</tr>
<tr>
<td>End $S_0$</td>
<td></td>
<td>${C_0}$ */ third CBGC */</td>
</tr>
</tbody>
</table>

| Total No of Local checkpoints generated = 3 |

5. CONCLUSION AND FUTURE WORK

In this paper we have proposed to use PET and MTBF of web services to decide whether checkpoints have to be taken at service calling locations. We have used Euclidean distances method to find out similar cases for the given web service execution instance and use them to estimate the execution time of the instance. If this estimated execution time of the called web service is less than its MTBF then there is no need to checkpoint its calling web service.

Recovery of applications based on checkpointing policies has been well studied in database and distributed computing fields. Due to lack of space we are not describing the implementation of recovery policy for web services. We intend to cover it in our future work.

When web services are orchestrated each service call results in creation of a new service instance and when the called service sends a reply back that service instance is destroyed. But if web services are choreographed service calls may be directed to already existing service instances. Also, when the called service sends a message to the caller, it might be in the middle of an operation, expecting some communication from the caller. Our previous checkpointing policy does not suffice to such a scenario. Hence we propose to develop a new checkpointing policy for Choreographed services as part of our ongoing research work.

6. ACKNOWLEDGEMENTS

I would like to express my gratitude to Prof. Hrushikesha Mohanty, Dept of Computer Science, University of Hyderabad, for his valuable suggestions that greatly helped me in writing this paper. I wish to extend my sincere thanks to him for sparing his precious time and encouraging me through out. I would like to wholeheartedly thank the management of CVR College of Engineering for providing me facilities to pursue my Ph.D. at University of Hyderabad.

7. REFERENCES


[8] [Sagnika Sen, Haluk Demirkan and Michael Goul.Towards a Verifiable Checkpointing Scheme for Agent-based Interorganizational Workflow System Docking Station Standards.