# Towards a Reliable Distributed Web Service Execution Engine

Xinfeng Ye
*Department of Computer Science, Auckland University, New Zealand*
*xinfeng@cs.auckland.ac.nz*

## Abstract

*This paper presents an approach for providing a reliable distributed Web Service Execution Engine. Instead of using a Web Service Execution Engine running on a single host to conduct the execution of a composite service, the responsibility of conducting the execution of the composite service has been delegated to the service providers that are chosen to provide the main functionalities of the tasks in the composite service. The operations of the tasks are represented as a set of XML-based notations. The notations are platform independent. Thus, they can be (a) retrieved by service providers based on different platforms, and, (b) interpreted and executed by the service providers. The approach also provides a mechanism for coping with possible failure in the system.*

## 1. Introduction

Web services are the fundamental building blocks for constructing distributed systems on the Internet. A composite service can be built from existing Web Services. A *composite service* consists of many tasks. The main functionalities of the tasks are provided by Web Services. For each task, apart from accessing a Web Service, users can also specify the operations for manipulating the information retrieved from the Web Service.

The execution of a composite service is conducted by a Web Service Execution Engine (WSEE). The WSEE conducts the execution of the composite service by invoking the relevant Web Services and manipulating the data received from the Web Services according to the operations specified for the tasks. Normally, the WSEE is hosted on a single site. As the WSEE needs to (a) interact with the Web Services that provide the functionality of the composite service and (b) carry out the required processing, it might become a *hot spot* in the system in terms of communication and computation. Research in mobile agent technology [3, 9, 14] has used the "moving code to data" concept to alleviate the hot spot problem mentioned above.

A mobile agent is a software component that has the ability to move from one host to another. A mobile agent carries both code and state. It can migrate to a host where the information to be manipulated resides. Then, the mobile agent manipulates the information at the host according to its code. Since the information no longer needs to be sent to a remote host, e.g. a WSEE, the network traffic can be reduced. A mobile agent moves amongst the hosts and carries out operations on the hosts that it migrates to. Thus, the computation has been distributed across the system. Hence, using the mobile agent technology can help to balance the load across the system.

This paper discusses an approach for providing a reliable distributed Web Service Execution Engine. Although it does not use mobile agents, it uses the "moving code to data" concept in conducting the execution of a composite service to address the hot spot problem facing the traditional WSEE. In this paper, instead of using a WSEE running on a single host to conduct the execution of a composite service, the responsibility of conducting the execution of the composite service has been delegated to the service providers that are chosen to provide the main functionalities of the tasks in the composite service. For a service provider that is chosen to execute a task, apart from providing the service requested by the task, the service provider is also responsible for executing the task's operations relating to the manipulation of data and the invocation of other tasks. The operations of the tasks are represented as a set of XML-based notations. The notations are platform independent. Thus, they can be (a) retrieved by service providers based on different technology, and, (b) interpreted and executed by the service providers. A mechanism for coping with possible failure in the system is also provided.

From the users' point of view, the approach in this paper should be attractive, since it should reduce the communication costs as well as the users' total reliance on a particular service provider for hosting WSEE. The approach in this paper should also be attractive to the service providers, since the service providers gain a new opportunity to collect revenue by hosting the distributed WSEE as a service.

## 2. Terminologies

Within a composite service, the tasks can be specified to run sequentially, concurrently or conditionally. If the execution of a task, say *B*, cannot start until another task, say *A*, terminates, *A* is called the *predecessor* of *B*, and, *B* is called the *successor* of *A*. For example, in Figure 1, *T2* is the successor of *T1*, and, *T1* is the *predecessor* of *T2*. A successor task is invoked by its predecessor. The invocation of some tasks might depend on the execution outcomes of their predecessors. For example, in Figure 1, *T2* decides whether *T3* or *T4* will be activated according to the operations specified by the user in *T2*. *T2* is called a *conditional branch task*, since only one of *T2*'s successors will be activated in a run.

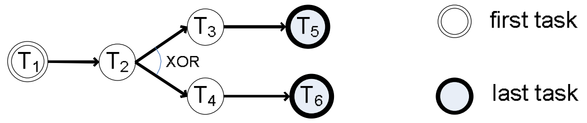

Figure 1 A Composite Service

It is assumed that a task in a composite service only accesses the operations provided by one Web Service[1]. For each task, it is assumed that multiple service providers are available for providing the services needed by the task. The user will choose one of the service providers to carry out the execution of the task while the other tasks are added to a candidate list for fault tolerance purpose. The service provider chosen to provide the service is called the *primary provider* while the ones on the candidate list are called *candidate providers*. For each task, the primary provider and the candidate providers form a group called *service provider group* of the task.

## 3. A Distributed Web Service Execution Engine

### 3.1. An Overview of the System

Figure 2 shows the interactions between the entities in the system. A Web Service Composition Tool (WSCT) is developed for specifying a composite service. A task description language is used to describe the operations to be carried out by the tasks in the composite service. Users can use the WSCT to locate service providers for each of the tasks in a composite service. Once the user completes the composition of a composite service, for each task in the composite service, the WSCT converts the operations of the task to a set of XML-based notations and stores the notations in a file, called *task file*. It is assumed that there exist some service providers, called the *task code repository* (TCR), that provide the service of hosting the task files. The WSCT dispatches the task files to the TCRs for storage. A task file is stored on several TCRs for fault tolerance purpose. The user can invoke a composite service using the WSCT. In turn, the WSCT instructs the primary provider of the first task in the composite service to retrieve the task file of the task and execute the task.

In order to execute a task, a service provider needs to have an interpreter to interpret and execute the operations stored in the task file. The service providers, which have the ability to run tasks, offer this ability as a service. They have a web operation for accepting the invocation calls for tasks. When a task is activated, the primary provider of the task retrieves the task file from a TCR and uses the interpreter to execute the task.

The task file of a task includes the operations for invoking the successor of the task. Thus, the WSCT only needs to activate the first task in a composite service. The other tasks will be activated by the service providers of the tasks' predecessors when their predecessors are executed.

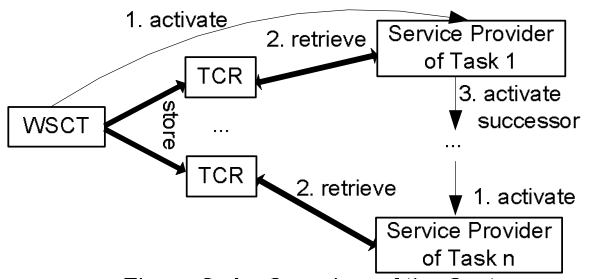

Figure 2 An Overview of the System

### 3.2. The Web Service Composition Tool

The Web Service Composition Tool (WSCT) is based on the light-weight visual tool for Web Service Orchestration [18]. The tool in [18] allows users to interact with a service registry to browse available Web Services that can be used for composing a composite service. Then, the user can use the tool to specify the service composition using the available Web Services. [18] only provides limited capability in describing the processing of the data retrieved from the Web Services. WSCT extends the work in [18] by introducing the Task Description Language (TDL) that

[1] If a task needs to access the operations of multiple Web Services, the task can be broken into several tasks.

allows the data retrieved from the Web Services being manipulated easily. Apart from specifying the primary provider for a task, WSCT allows users to provide a list of candidate providers for carrying out the task to cope with possible failures in the system. The user assigns ranks to the candidate providers according to the user's preference.

## 3.3. The Task Description Language (TDL)

The Task Description Language (TDL) is used for describing the operations of a task. It is based on Java, albeit much simple than Java. TDL supports sequential, while loop, and, if statements as well as some arithmetic, relational and logical operators, e.g. +, |, >, etc. It also allows users to use primitive data types, e.g. *int*, *bool*, etc. Users can also define objects.

A task is defined like a procedure with a list of input parameters as shown in the example below. In the example, the name of the task is *T*. It accepts two input parameters, *Para1* and *Para2*, of type *ParaType1* and *ParaType2*[2].

```
T(ParaType1 Para1, ParaType2 Para2) {
  ... // operations of the task
}
```

In TDL, a call to a Web Service is represented like a procedure call as shown in the following example.

*Record r = WSCall(WS_URL, Op, ParameterList)*

In the above example, *Record* is a user-defined class type that corresponds to the result returned from the call to the Web Service. *WS_URL* points to the location where the WSDL file of the Web Service is stored. *Op* is the name of the web operation to be called. The parameters to be passed to the Web Service are concatenated to form a string with ";" as the delimiter separating the parameters[3]. The string is denoted as *ParameterList* in the above example. The interpreter for executing the task will generate the client-side stub for accessing the Web Service according to the WSDL file given at *WS_URL*. The interpreter extracts the parameters from the *ParameterList* and carries out the call to the Web Service through the client-side stub.

Once a user completes the specification of a composite service, for each task, the WSCT converts the TDL descriptions of the task to a set of XML-based notations and stores them in a task file. Each task is given a unique name that is used as the title of the file. A task file is partitioned into two parts. One part contains the operations of the task. The other part contains the variables referenced in the task's operations. The example below shows converting a TDL code fragment into XML-based notations. Assume that the *if* statement is the last operation carried out by *T2* in Figure 1. Depending on the values of *a* and *b*, either *T3* or *T4* will be activated next. The activation of a task is through calling the Web Service that will host the execution of the task. Assume that the service providers of *T3* and *T4* provide web operation "*activateTask*" for task activation purpose. The parameters to be passed to the *activateTask* operation should include a list of locations from where the task file of the activated task can be retrieved, the ID of the task to be activated, and, the data to be passed to the task. In this example, it is assumed that (a) the IDs of the tasks are *T3* and *T4* respectively, (b) the task files can be downloaded at *tcr.com/T3.xml* and *tcr.com/T4.xml*[4], and, (c) there are no data to be passed to *T3* and *T4* when they are invoked. Thus, *ParameterList* in the call to *T3*'s and *T4*'s service providers are "*tcr.com/T3.xml;T3;*" and "*tcr.com/T4.xml;T4;*" respectively. It should be noted that ";" delimits the parameters to be passed to the *activateTask* method; and, the last ";" in *ParameterList* is followed by an empty string since no data are passed to *T3* and *T4* in this example.

```
if (a < b) {
   WSCall("provider1.com/wsee.asmx?WSDL",
          "activateTask", "tcr.com/T3.xml;T3;");
}
else {
  WSCall("provider2.com/wsee.asmx?WSDL",
          "activateTask", "tcr.com/T4.xml;T4;");
}
```

The XML-based notations representing the above TDL fragment are as below. The notations for the *else* branch of the *if* statement have been omitted, since they are similar to the *then* branch. It can be seen that XML tags, e.g. *if*, *condition*, *then*, *wscall*, etc., are used to represent the meaning of the operations. Tag *var* represents a variable.

```
<if>
  <condition>
    <operator opName="lt">
      <parameter>
        <var varName="a"/>
      </parameter>
      < parameter>
        <var varName="b"/>
      </parameter>
```

[2] If a task is the joining point of several predecessor tasks, each predecessor task will set the values of the parameters relevant to it and set the irrelevant ones to a special null value. The task should include the code for retrieving the values passed in by its predecessors from the parameters.

[3] Although object can be represented in TDL, the prototype implementation only supports string type parameters.

[4] For simplicity, this example assumes that the task files are only stored on one TCR.

```
    </operator>
  </condition>
  <then>
    <wscall>
        <parameter>
          provider1.com/wsee.asmx?WSDL
        </parameter>
        <parameter>
          activateTask
        </parameter>
        <parameter>
          tcr.com/T3.xml;T3;
        </parameter>
     </wscall>
   </then>
   <else>
     ...// omitted
   </else>
</if>
```

A task file can be represented as a tree. To execute the operations of a task, the interpreter carries out a traversal of the tree (with some nodes being skipped in the conditional branch case). Depending on the meaning of the tags, the interpreter executes the operations accordingly. For variables, the interpreter needs to search the tree to locate the values for the variables[5].

## 3.4. Code Pre-fetching

For the discussion up to now, it is assumed that the service provider of a task retrieves the task file from a TCR when the task is activated. Thus, the execution of the task would not start until the service provider receives the task file. This introduces a delay in the execution of the task. It is possible to eliminate this delay in some cases through pre-fetching the task file.

If a task, say *T*, is not a conditional branch task in a composite service, when *T* is activated, the service provider of *T* can also inform the service provider of *T*'s successor to retrieve the task file of *T*'s successor. For example, in Figure 1, when *T1* is activated, the service provider of *T1* informs the service provider of *T2* to retrieve *T2*'s task file from a TCR. Thus, the execution of *T1* and the retrieving of *T2*'s task file are carried out simultaneously. If the time for retrieving *T2*'s task file is shorter than executing *T1*, when *T2* is activated later, *T2*'s task file is already available on the service provider of *T2*. As a result, *T2* can be executed immediately without any delay.

If a task is a conditional branch task, e.g. *T2* in Figure 1, it is impossible to determine which task will be executed next until the execution of *T2* completes. However, it is possible to ask the service providers of all the successors of the conditional branch task to retrieve the successors' task files. For example, in Figure 1, when *T2* is activated, the service providers of *T3* and *T4* are both informed to retrieve the task file of *T3* and *T4* respectively. When *T2* is completed, either *T3* or *T4* will be activated. Since the code of *T3* and *T4* has already been retrieved by their respective service providers, if they are activated, their execution can be started immediately. Of course, this approach is a trade-off between the execution efficiency and the communication cost. This is because the task file of the task that is not activated will be discarded. Thus, the effort of retrieving the file is wasted.

In our prototype implementation, the service provider of a task, say *T,* asks its successor pre-fetch the successor's task file only if *T* is not a conditional branch task. When a user completes the composition of a composite service, the WSCT checks the type of each task in the composite service. If a task, say *T*, is not a conditional branch task, the WSCT inserts some instructions into *T*'s task file. The instructions ask the service provider of *T*'s successor pre-fetch *T*'s successor's task file.

## 3.5. Reliable Execution

The successful execution of a composite service depends on the successful execution of each of the tasks in the composite service. Replication has been widely used to achieve reliability in many systems [17]. To make their service reliable, service providers can replicate their services to mask failure and make failure transparent to the users [1, 19]. However, there is still a (small) possibility that the mechanisms deployed by a service provider fail to handle the failures of the system adequately. As a result, the service provider fails to deliver the service being advertised. This section discusses how to cope with this type of failure.

It is assumed that multiple service providers are available for carrying out a task in a composite service. Thus, we can always find a service provider to take over from the failed service provider. The mechanisms used in this paper achieve two objectives, i.e. non-blocking and exactly-once semantics. *Non-blocking* means the failures of a service provider does not block the progress of the computation of a composite service. *Exactly-once semantics* means that the effects of executing a task are only made permanent on one service provider. To make the mechanisms work, for each task *T*, the WSCT adds some information to *T*'s task file to make *T*'s service providers know the

[5] TDL does not support variable scoping. Thus, a variable can only be defined and used once in a task.

identities of all the service providers of *T*'s successor task and the identity of the primary provider of *T*'s predecessor task.

**3.5.1. Non-blocking** The key to non-blocking is to ensure that the failure of a service provider that is executing a task can be detected. Since each task has a group of service providers, to satisfy the non-blocking requirement, the candidate providers need to monitor the failure of the primary provider. When the primary provider fails, the candidate providers run an election protocol [6] to elect the next highest ranked candidate provider as the new primary provider. The new primary provider will re-execute the task. Thus, the failure of a service provider will not block the progress of a composite service.

To allow the monitoring of the primary provider, when the primary provider of a task is activated, it informs the candidate providers in the task's service provider group and passes the data that it received from the predecessor task to the candidate providers in case they need to carry out the task if the primary provider fails. The primary provider will periodically send alive-message to the candidate providers. When a candidate provider suspects that the primary provider has failed, it initiates an election algorithm to choose a new primary provider.

The other issue that needs to be considered is how to ensure the successful invocation of a task. A successful invocation of a task means two conditions are satisfied (a) the primary provider of the task starts executing the task, and, (b) the candidate providers start monitoring the primary provider. Condition (b) needs to be satisfied, since it is necessary to discover the failure of the primary provider during the execution of the task. To ensure that condition (b) can be satisfied, the candidate providers need to acknowledge the primary provider when they are informed about the invocation of the task. Once condition (b) is satisfied, the primary provider informs its predecessor. In turn, the predecessor task terminates. That is, the primary provider of the predecessor task discards the predecessor task's file and the candidate providers of the predecessor task stop monitoring the primary provider. This is because the computation has successfully progressed from the predecessor task to its successor. On the other hand, if the primary provider of a task does not inform its predecessor that it has been activated successfully, the predecessor will regard the primary provider as failed. The predecessor will activate the highest ranked candidate provider to execute the task.

**3.5.2. Exactly-once Semantics** [16] points out that in the Internet, it is impossible to detect the failure of the system reliably due to the unpredictable communication delays. Thus, during its execution, a primary provider might be wrongly suspected of having failed. As a result, a new primary provider is elected and carries out the task. Hence, the task is executed twice (i.e. by the original primary and the new primary providers respectively). Similar problem occurs if a service provider is wrongly suspected of not being activated successfully, since the predecessor will activate another service provider.

For idempotent operations (i.e. the operations without side effects), multiple execution of a task does not create any problem. The effects of many non-idempotent operations can be undone by executing compensating operations. For example, the effect of reserving a flight ticket can be undone by executing a reservation cancellation operation. In this paper, it is assumed that the effects of all the non-idempotent operations can be undone by executing compensating operations. This means that the service providers also provide operations to undo the effect of each web operation being offered. Thus, to solve the problem of multiple execution of a task caused by incorrectly suspecting a primary provider of having failed, a scheme is needed to (a) detect the occurrence of multiple execution of a task, and, (b) undo the effects caused by the multiple execution of the task.

The mechanism used in this paper detects multiple execution of a task at task activation time. **Rule 1** and **2** of the mechanism handle the multiple execution of a predecessor task while **Rule 3** deals with the multiple invocation of a task.

In the absence of a failure, a task is to be executed by its primary provider. Thus, the primary provider, say *pp*, will activate the successor task. The service providers of the successor task regard *pp* as the *default activator*.

**Rule 1**: If the service provider of a task, say *T*, receives another activation call from service provider *sp* after receiving the activation call from the default activator, *sp* will be notified that an activation call has been received from the default activator. As a result, *sp* will undo the operations of the task.

Figure 3 depicts the problem being handled by **Rule 1**. SG1 and SG2 are the service provider groups of two tasks. Each group consists of three service providers. Original, the primary provider of SG1 is $sp_{11}$. Thus, all the service providers in SG2 regard $sp_{11}$ as the default activator. Figure 3(a) shows that $sp_{11}$ has completed its task and sends an activation call to $sp_{21}$. In the meantime, $sp_{11}$ is incorrectly suspected by $sp_{12}$ and $sp_{13}$ of having failed. As a result, $sp_{12}$ and $sp_{13}$ run the election algorithm to choose $sp_{12}$ as the new primary provider for SG1. Figure 3(b) shows that $sp_{12}$ has completed the execution of the task and sends an activation call to $sp_{21}$. Since $sp_{21}$ has already received an activation call from $sp_{11}$, Figure 3(c) shows $sp_{21}$ informs $sp_{12}$ that $sp_{11}$ is the default activator. Thus, $sp_{12}$ will undo the task.

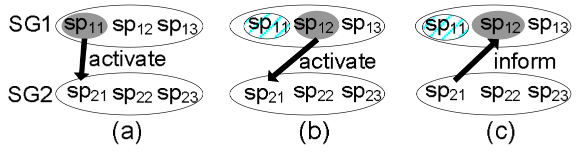

Figure 3 Multiple Execution of Task (I)

**Rule 2**: If the service provider of a task *T* receives an activation call from a provider that is not the default activator[6], the service provider initiates an agreement protocol amongst *T*'s service providers. The agreement protocol determines the identity of the new default activator. If multiple service providers have made the activation call, the provider whose activation call is received first will become the new default activator[7]. The service providers that made the activation calls are notified of the identities of the new default activator. If a service provider does not become the default activator, it will undo the operations of its task.

Figure 4 shows the scenario being dealt with by **Rule 2**. Figure 4(a) shows that $sp_{12}$ and $sp_{13}$ wrongly suspect that $sp_{11}$ has failed and $sp_{12}$ becomes the new primary provider of SG1. $sp_{12}$ sends out the activation call on completion of the task. $sp_{12}$'s activation call is received by $sp_{21}$ before $sp_{11}$'s activation call arrives. Since $sp_{12}$ is not the default activator, the providers in SG2 run an agreement protocol to agree on having $sp_{12}$ as the new default activator and inform $sp_{12}$ as shown in Figure 4(b). Figure 4(c) shows that the activation call from $sp_{11}$ finally arrives at $sp_{21}$. Since $sp_{11}$ is no longer the default activator, $sp_{11}$ is in the same situation as $sp_{12}$ in Figure 3(b). Thus, $sp_{11}$ will undo the operations of the task.

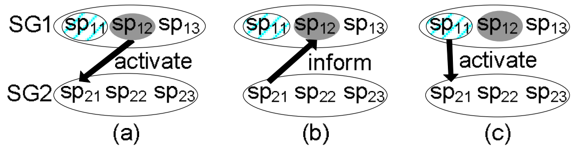

Figure 4 Multiple Execution of Task (II)

**Rule 3**: If a candidate provider of a task *T* receives an activation message from the default activator[8], the candidate provider initiates an agreement protocol amongst *T*'s service providers. The agreement protocol determines who will carry out the task being activated.

Figure 5 illustrates the problem addressed by **Rule 3**. In Figure 5(a), $sp_{21}$ is activated by $sp_{11}$. Figure 5(b) shows that $sp_{11}$ wrongly suspects $sp_{21}$ of having failed as $sp_{11}$ does not receive the reply from $sp_{21}$ in time. Thus, $sp_{11}$ activates $sp_{22}$. $sp_{22}$ initiates the agreement protocol. Since $sp_{21}$ has started executing the task, the service providers in SG2 will agree that $sp_{21}$ will still be responsible for executing the task.

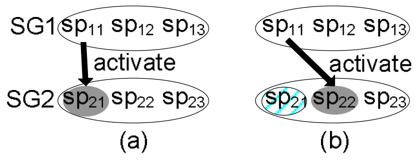

Figure 5 Multiple Activation of the Task

The prototype implementation of the system uses the agreement protocol described in [5] that has a relatively low overhead. It allows an agreement to be reached as long as the majority of the service providers are alive.

The last task in a composite service does not have a successor. Thus, we cannot rely on the mechanism described above to ensure exactly-once semantics. If a task *T* is the last task, the primary provider initiates an agreement protocol amongst *T*'s service providers when it completes the execution of the task. The agreement protocol makes the service providers agree on who is responsible for executing the task. This is to ensure that only one provider makes the effects of the task permanent.

The WSCT inserts the necessary instructions for carrying out the mechanisms that ensure non-blocking and exactly-once semantics into the task file of each task. Thus, users do not need to consider these issues when they write the tasks.

## 3.6. Web Services Supports

To host a distributed execution engine, a service provider needs to implement a service interface that contains several web operations. The most important operations are explained below.

*activateTask* is called when a task is activated. This operation has three parameters, i.e. a list of locations of the task file on the TCRs, the unique name assigned to the task, and, a list containing the parameters to be passed to the task. *activateTask* checks whether the file of the specified task has already been fetched from the

[6] This is the case that another service provider has executed *T*'s predecessor task.

[7] The original default activator can still become the new activator if its activation call is received first.

[8] This is the case that the predecessor suspects that the primary provider has failed and invokes the task on a candidate provider.

TCR[9]. If the task file is not available, the service provider will first fetch the task file from the TCR before starts executing the task. *activateTask* needs to implement the interpreter that executes the operations in the task file.

*prefetchCode* is used to inform a service provider to pre-fetch the task file of the specified task. This operation has two parameters, i.e. a list of locations of the task file and the unique name assigned to the task.

*activationAck* operation is called by the service provider of a task *T* to inform the service provider of *T*'s predecessor task about whether *T* has been activated successfully. The operation has two parameters. One is the flag indicating whether the activation is successful; and, the other shows the identity of the default activator of *T*. If the service provider whose *activationAck* operation is called is not the default activator, the service provider has to undo the operation of the task as discussed in §3.5.2.

Each service provider also needs to provide some web operations to allow the execution of the leader election algorithm and the agreement protocol as well as the operations relating to failure monitoring to counter possible failures.

## 4. Related Work

Research in Mobile Web Services focus on integrating mobile agents with Web Services. [7] proposes a framework for Mobile Web Services. In the framework, mobile agents carry the execution plan and the code for processing data retrieved from Web Services. As an agent needs to carry the complete execution plan of a composite service and the code for interacting with all the Web Services used in the plan, the communication overhead of agent migration need to be carefully balanced with the gains obtained by having the agent processing the data at the service providers locally. The framework also does not address the handling of failures. For the approach in this paper, a service provider only retrieves the code of the task that it is responsible for. Hence, it does not cause any unnecessary communication overhead.

[12] describes an approach of using mobile agent to carry out the execution of a composite service. It also uses mobile agents to select the service providers for executing the tasks in a composite service. As [7], a mobile agent also needs to carry the complete execution plan of the composite service. Although [12] mentioned that reliability of execution can be achieved through a backup approach, it did not address the issues like agent lost and duplicated executions of tasks.

[13] presents an architecture that allows Web Services to be transferred to mobile devices and execute there. The system is built on Aglets [10]. Thus, it requires a Java-based environment to run the service. In contrast, the approach described in this paper represents the operations of the tasks using a set of XML-based notations. Thus, as long as the service providers have interpreters that understand the notations, the operations of the tasks can be carried out on platforms that are based on different technology.

BPEL4WS is an XML based Web Service composition language. It is used to compose multiple web services into an end-to-end business process. As BPEL4WS is designed for Web Services choreography, it is not always easy to describe some operations to manipulate data in BPEL4WS. [11] describes an approach that overcomes this limitation of BPEL4WS. [11] enables Web Services to accept and run mobile code. In [11], the code of a task is first translated into an XML-based language called X#. The X# code can then be deployed to the service containers. The X# code is translate back to a platform specific language at the service containers and installed there. The installed code can be activated through a normal Web Service call. The focus of [11] is on developing a platform-neutral language to represent the code of a task. Our approach not only addresses the issue of representing the operations of a task in a platform-neutral way, it also discusses the execution of a composite service and the techniques for handling failures.

Although the approach in this paper does not use mobile agents, it uses the same principle as mobile agents in terms of sending code to the locations where the data reside. Many approaches to make mobile agents tolerate failures have been proposed. Some of the approaches, e.g. [4] and [8], are based on the assumption that there is a perfect failure detector. However, in the Internet, the upper-bound of the network delay and service delay are not known in advance. Therefore, failure detectors are unreliable, i.e. they might declare the failure of a service incorrectly. [15] proposes a fault tolerant mobile agent system that prevents blocking (when the majority of the machines in the system are alive) and detects duplicated execution. It is based on agent replication and it requires all the agents run an agreement protocol when they complete a task. Running an agreement protocol is an expensive operation in terms of the synchronizations being carried out amongst the participating agents. In contrast, the approach in this paper only requires the service providers to run the agreement protocol when there are possible failures or

[9] The task file might have not been pre-fetched if the predecessor of the task is a conditional branch task.

duplicated execution in the system. Thus, the overheads caused by the operations for tolerating failures have been minimized.

## 5. Conclusions and Future Work

This paper describes an approach for providing a distributed Web Service Execution Engine. Users use a graphical tool to carry out service composition as well as describing the operations of the tasks in a composite service. The operations of each task are converted to a set of XML-based notations. As a result, the tasks can be executed on platforms using different technologies. In contrast to the mobile agent based approach, instead of sending the operations of all the tasks in a composite service to each service provider, a service provider in this paper only needs to obtain the task that the service provider has been selected to execute. Thus, the approach in this paper has lower communication overhead when transmitting the operations to a service provider. The fault tolerant mechanism in this paper uses a demand-driven approach. It only runs the expensive agreement protocols when there is a (possible) failure in the system. Thus, in the absence of system failure, the overhead of running the fault tolerant mechanisms is kept low.

The biggest obstacle to the adoption of the approach in this paper seems to be persuading the service providers to provide the interpreter used to execute the tasks. However, hosting the WSEE is provided as a service. This means it provides an opportunity for the service providers to generate revenue by offering the service. Hopefully, this financial incentive will convince the service providers to adopt the approach discussed in this paper.

We are currently working on integrating a service broker into the WSCT. The broker uses a scheme similar to [2] and [20] to select the service provider for each task in a composite service to maximize the QoS of the composite service. With the help of the broker, users do not need to manually select service providers. We are also evaluating how the system affects the network load under various application scenarios.

**Acknowledgment.** We would like to thank the anonymous reviewers for their useful comments.

## References

[1] N. Aghdaie, Y. Tamir, Fast Transparent Failover for Reliable Web Service, Proc. of Intl Conf. on Parallel and Distributed Computing and Systems, 2003

[2] D. Ardagna, B. Pernici. "Global and Local QoS Constraints Guarantee in Web Service Selection," IEEE International Conference on Web Services, 2005

[3] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? IBM Research report, RC19887, 1994

[4] P. Dasgupta, Fault tolerance in MAgNET: A mobile agent e-commerce system. In Proc. of the 6th Int. Conference on Internet Computing, 2000

[5] X. D´efago, A. Schiper, and N. Sergent, Semi-passive replication. In Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98), 43–50. 1998

[6] H. Garcia-Molina, "Elections in a Distributed Computing System", IEEE Transactions on Computers 31(1), pp.48-59, 1982

[7] F. Ishikawa, N. Yoshioka, Y. Tahara, and S. Honiden, "Towards Synthesis of Web Services and Mobile Agents." In AAMAS2004 Workshop on Web Services and Agent-based Engineering, 2004.

[8] D. Johansen, K. Marzullo, F. Schneider, K. Jacobsen, and D. Zagorodnov, NAP: Practical fault-tolerance for itinerant computations. In Proc. of 19th IEEE Int. Conference on Distributed Computing Systems, 1999

[9] D. Kotz, R. Gray, and D. Rus, "Future Directions for Mobile-Agent Research," IEEE Distributed Systems Online, 3(8), Aug. 2002.

[10] D. Lange, M. Oshima, G. Karjoth, and K. Kosaka, "Aglets: Programming Mobile Agents in Java," In Proc. Worldwide Computing and its Applications, 1997.

[11] P. Liu and M. Lewis, Mobile Code Enabled Web Services, IEEE International Conference on Web Services, pp. 167-174, 2005.

[12] Z. Maamar, Q. Z. Sheng, and B. Benatallah. "Interleaving web services composition and execution using software agents and delegation." In AAMAS2003 Workshop on Web Services and Agent-based Engineering, 2003.

[13] Z. Maamar, Q.Z. Sheng, and B. Benatallah, "On Composite Web Services Provisioning in an Environment of Fixed and Mobile Computing Resources", Information Technology and Managemenl, Kluwar Academic, 2004, pp 251-270. Vol 5, No 3-4

[14] D. Milojicic. Mobile agent applications. IEEE Concurrency, 7(3):7–13, 1999

[15] S. Pleisch, and A. Schiper, Fault-tolerant mobile agent execution. IEEE Trans. Comput. 52, 2 (Feb.), 209–222. 2003

[16] S. Pleisch, A. Schiper: Approaches to fault-tolerant and transactional mobile agent execution -- an algorithmic view. ACM Comput. Surv. 36(3): 219-262, 2004

[17] Y. Saito, and M. Shapiro, Optimistic replication, ACM Comput. Surv. , 37 (1 ): p. 42-81, 2005

[18] H Wan, Building Dynamic Inter-organization Business Processes by Orchestrating Web Services, MSc Thesis, Dept. of Computer Science, Auckland University, 2005

[19] X. Ye, and Y. Shen: A Middleware for Replicated Web Services. IEEE International Conference on Web Services, pp. 631-638, 2005

[20] L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam, H. Chang. "QoS-Aware Middleware for Web Services Composition," IEEE Transactions on Software Engineering, vol. 30, no. 5, pp. 311-327, May, 2004.