Orchestrating Data-centric Workflows

Adam Barker*, Jon B. Weissman†*, and Jano van Hemert*
* National e-Science Centre (NeSC), University of Edinburgh, United Kingdom.
† Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA.

***Abstract*—When orchestrating data-centric workflows as are commonly found in the sciences, centralised servers can become a bottleneck to the performance of a workflow; output from service invocations are always transferred via a centralised orchestration engine, when they should be passed directly to where they are needed at the next service in the workflow. To address this performance bottleneck, this paper presents a lightweight Web services architecture and concrete API, based on a centralised control flow, distributed data flow model. Our architecture maintains the robustness and simplicity of centralised orchestration, but facilitates choreography by allowing services to exchange data directly with one another, reducing data that needs to be transferred through a centralised server. Furthermore, our architecture is a flexible, non-intrusive solution, as existing service definitions do not have to be altered prior to enactment.**

***Index Terms*—Systems architecture, workflow optimisation, Web services, decentralised orchestration.**

I. Introduction

Service-oriented architectures are a popular architectural paradigm for building software applications from a number of loosely coupled distributed services. Although the concept of service-oriented architectures is not a new one, this paradigm has seen wide spread adoption through the Web services approach, which has a suite of simple standards (XML, WSDL, SOAP etc.) to facilitate service interoperability.

Web services in their vanilla form provide a simple solution to a simple problem, things become more complex when a group of services need to coordinate together to achieve a shared task or goal. This coordination is often achieved through the use of workflow technologies. As defined by the Workflow Management Coalition [4], a workflow is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant (a resource; human or machine) to another for action, according to a set of procedural rules.

There are two common architectural approaches to implementing workflow; *service orchestration* and *service choreography*. Service orchestration refers to an executable business process that may interact with both internal and external services. Orchestration describes how services can interact at the message level, with an explicit definition of the *control flow* and *data flow*. By control flow we refer to typically short messages that are used to trigger state changes at the remote services, data flow on the other hand are mostly large data that are given to the receiving services for processing. Orchestrations can span multiple applications and/or organisations and result in long-lived, transactional processes, services themselves have no knowledge of their involvement in a higher level application. A central process always acts as a controller to the involved services, both control and data flow messages pass through this centralised server. The *Business Process Execution Language (BPEL)* [10] is an executable business process modelling language and currently the current de-facto standard way of orchestrating Web services. It has broad industrial support from companies such as IBM, Microsoft and Oracle.

Service choreography on the other hand is more collaborative in nature. A choreography model describes a collaboration between a collection of services in order to achieve a common goal. Choreography describes interactions from a global perspective, meaning that all participating services are treated equally, in a peer-to-peer fashion. Each party involved in the process describes the part they play in the interaction. Choreography focuses on message exchange, all involved services are aware of their partners and when to invoke operations. The *Web Services Choreography Description Language (WS-CDL)* [6] is an XML-based language that can be used to describe the common and collaborative observable behavior of multiple services that need to interact in order to achieve a shared goal. Currently this language is in the W3C candidate recommendation stage and there are no concrete implementations.

Orchestration differs from choreography in that it describes a process flow between services from the perspective of one participant (centralised control), choreography on the other hand tracks the sequence of messages involving multiple parties (decentralised control, no central server), where no one party truly owns the conversation.

A. Problem Statement and Paper Contributions

To illustrate our problem statement, it is best to consider a simple, artificial example. A scientist constructs a workflow to collect, analyse and visualise a distributed set of data. In this workflow three distributed databases are queried through Web service interfaces, the results of these queries are then forwarded to an analysis Web service, which computes some function of these data and outputs the results to a visualisation Web service, finally the output from the visualisation service is used as input to a program running on a user's terminal. As illustrated by the top of Figure 1 the output of the database queries (3Gb in total) are moved to the orchestration engine and forwarded as input to the analysis Web service, whose output (1GB) is again passed through the orchestration engine and finally used as input to the visualisation service. The visualisation service forwards the results of the workflow

(via the orchestration engine) directly to the user (500Mb). When output from one service invocation is directly (with no alteration) used as input to another, we label this *intermediate data*.

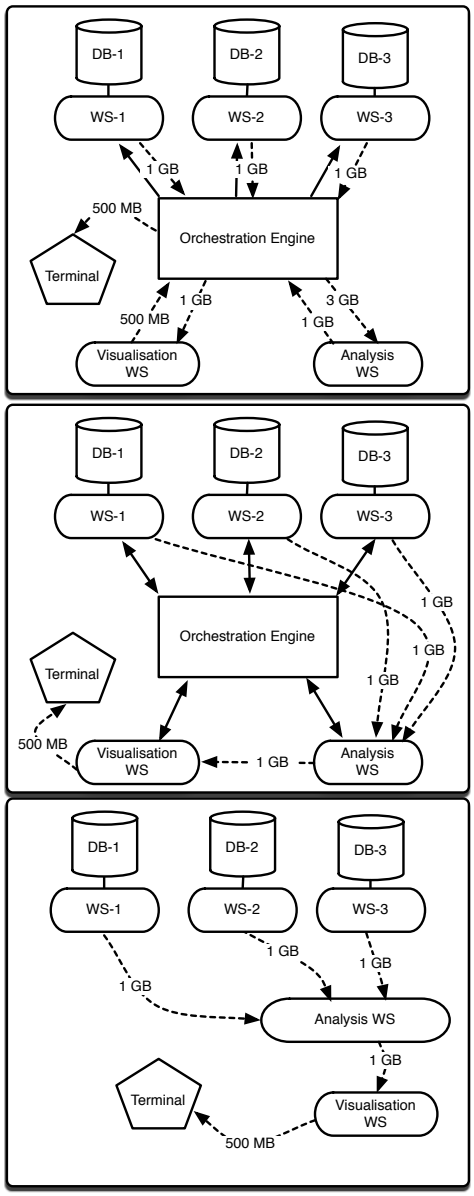

Fig. 1. Example scenario: Orchestration (top), Centralised control flow, distributed data flow (middle) and Choreography (bottom). Web services are represented by rounded rectangles, control flow messages by solid arrowed lines and data flow messages by dashed arrowed lines

Currently most research has focused on designing languages for and implementing service orchestrations, where both control and data flow pass through a centralised server. There are a plethora of orchestration frameworks which will automate these tasks for you, examples can be seen in the Business Process Modelling community through BPEL, life sciences through Taverna [13] and computational Grid community through Pegasus [2].

Centralised control through an orchestration engine is a valid solution for scenarios found in e-Commerce, where relatively small quantities of intermediate data are moved between services in a workflow. However, centralised servers make less sense when dealing with data centric workflows (Gbs/Tbs) as are commonly found in the sciences; for example constructing workflows which analyse data from the Sloan Digitial Sky Survey (SDSS) [9]. Passing large quantities of intermediate data through a centralised orchestration engine results in unnecessary data transfer, thereby decreasing the performance of a workflow. Furthermore, it may be the case that the groups of services interacting in the workflow are deployed near to one another and the orchestration engine controlling them is sitting on a user's desk on the other side of the world. If this is the case, the intermediate data has to make a large hop back to the orchestration engine when locality could be exploited by piping it directly to the next service in the workflow which happens to be local. Referring to our scenario again using orchestration a total data transfer of 9Gb of data is required to enact the workflow.

In the scientific domain, where large quantities of intermediate data are passed between services, a choreography model is a more efficient architecture. By adopting a choreography model, the output of a service invocation can be passed directly to where it is required as input to the next service, thereby decreasing intermediate data and as a result decreasing network traffic. This is illustrated by the bottom of Figure 1, by using a choreography model data is not being passed through a centralised server, the total data transfer to enact the workflow has been reduced to 4.5Gb, a total saving of 4.5Gb.

In practise, the design process and execution infrastructure for service choreography models are inherently more complex than orcehstration. Decentralised control brings a new set of problems which are the result of message passing between distributed concurrent processes. This paper proposes a novel hybrid architecture, based on *centralised control flow, distributed data flow* [7], illustrated by the middle section of Figure 1. This architecture sits between pure orchestration and pure choreography. A centralised orchestration engine issues control flow messages to Web services taking part in the workflow, however enrolled Web services can pass data flow messages amongst themselves, like a peer to peer model. This model maintains the robustness and simplicity of centralised orchestration but facilities choreography by avoiding the need to pass large quantities of intermediate data through a centralised server.

II. System Overview of Hybrid Architecture

This Section discusses why current standards alone are sufficient to support service orchestration but need to be adapted to support more complex models of interaction, like choreography. Our hybrid architecture is introduced which facilitates a centralised control flow, distributed data flow model. A concrete API is presented and applied to the simple example discussed in Section I-A.

A. Achieving Choreography

The Web Services Definition Language (WSDL) provides a standardised wrapper to expose application code to a network. This wrapper was deliberately designed to be simple and lightweight and is a contributing factor to the success of the Web services approach to service-oriented architectures. However, WSDL definitions alone cannot support the functionality required of choreography, where the output of a method invocation is sent directly to where it is required next in the workflow; instead the output is send back to the orchestration engine, application etc. which invoked it.

To enable Web services to pass data directly amongst themselves an extra layer of functionality needs to be added to the stack, this can be achieved in a number of ways:

• **Choreography interface:** To facilitate choreography an extra interface could be wrapped around existing service definitions. The current proposed standard, WS-CDL is designed to sit on top of WSDL; WSDL focuses on capturing message types while WS-CDL is about capturing behaviour. A user models a choreography from a global perspective, each individual service will have to be programmed by a developer in such a way that they talk to one another, and in doing so, enforce the constraints of the choreography. This model allows services to exchange data with one another with no centralised server, however vanilla Web services can not take part in the choreography without the added WS-CDL layer and choreography specific programming.

• **Installation of proxies:** An alternative approach to achieving choreography is the installation of a proxy which sits infront of a service or groups of services. A proxy provides an extra set of functionality (e.g. passing output directly to where it is needed) and invokes a service on an application's behalf. Communication between the proxy and services can be local if they are installed on the same Web server or domain.

Although a choreography interface is the optimal solution in terms of performance, this extra interface is intrusive to the services themselves and means that each individual service definition must be wrapped with a choreography interface prior to workflow enactment. Web services are owned and maintained by different organisations and may not agree to installing specialist interfaces in order to facilitate choreography, unless they have something to gain. Proxies are less intrusive and offer an advantage as they are entirely external to the Web services themselves and services require no alteration or knowledge that they are taking part in an interaction.

B. Proxy Architecture

In order to provide Web services with the required functionality to realise a centralised control flow, distributed data flow model, this paper presents a proxy architecture. A proxy is a lightweight, non intrusive piece of middleware which provides a gateway and standard API to Web service invocation. A proxy allows Web services to exchange data flow messages directly with one another (avoiding transferring them through a centralised server). Proxies are installed as 'closely' as possible to enrolled Web services, by close we mean preferably

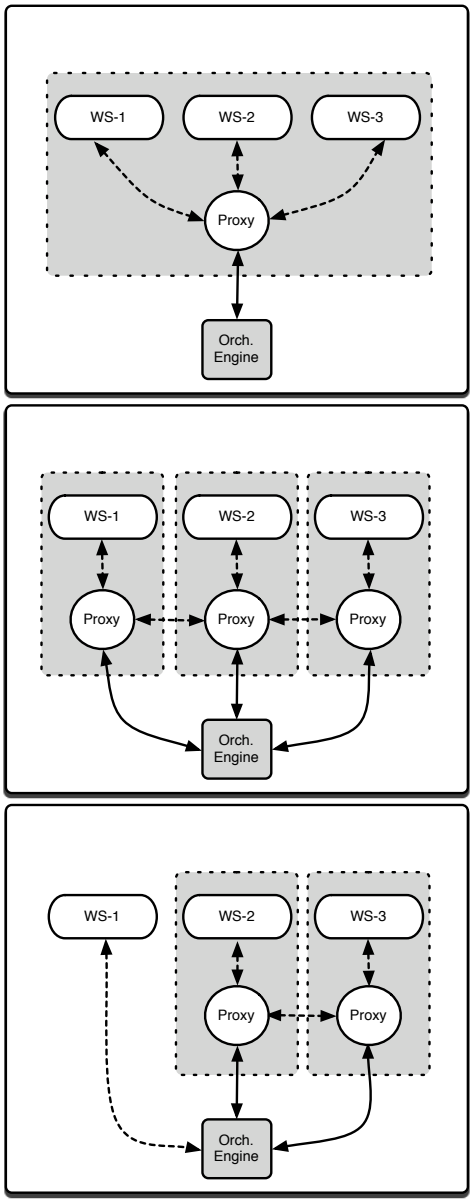

Fig. 2. One proxy, multiple services, 1:N (top), One proxy, one service, 1:1 (middle), mixed components (bottom). Proxies are represented by circles

on the same Web server or domain so that communication between a proxy and a Web service goes over a local, not Wide Area network. Depending on the preference of an administrator, a proxy can be responsible for one Web service, 1:1 or many Web services, 1:N; this is represented by the top and middle of Figure 2.

Proxies themselves are exposed through a WSDL interface, allowing them to be built into workflows or higher level applications like any other Web service. As everything is exposed through a WSDL interface, this means that workflows can be made up with a combination of proxies and vanilla Web services, represented by the bottom of Figure 2.

As discussed earlier, our hybrid architecture sits between

a purely centralised solution (orchestration) and a purely decentralised solution (choreography). To realise a centralised control flow, distributed data flow model, proxies are controlled by a centralised orchestration engine which is executing an arbitrary workflow language. However, only control flow messages are passed through the orchestration engine, larger data flow messages are exchanged between proxies in a peer to peer fashion, unless a proxy is explicitly told to do otherwise. Proxies exchange references to the data with the orchestration engine and pass the real data directly to where it is needed for the next service invocation; this allows the orchestration engine to monitor the progress and make changes to the execution of a workflow.

C. Proxy API

Proxies are designed to be installed on the same Web server or domain as a number of data centric Web services. The proxy architecture is available as an open-source toolkit implemented using a combination of Java and the Apache Axis Web services toolkit [11]. They are extremely simple to install and configure and can be dropped into a AXIS container running on a Tomcat server and configured remotely, no specialised programming needs to take place in order to exploit the functionality. The architecture is multi-threaded and allows several applications to invoke methods concurrently. A proxy has a thread pool and when that thread pool is full the request is placed on an input queue and dealt with in First In First Out (FIFO) order. Results from Web service invocations are stored at a proxy in a Hashtable which maps a UID reference to a data element. Proxies are made available through a standard WSDL interface, the Java representation of that interface is displayed in Figure 3, all methods are invoked by an orchestration engine except `stage`. The proxy has the following methods:

• **invoke** is the main proxy method and provides a gateway to Web service invocation. This method takes as input details of the Web service to be invoked, including the location of a WSDL, portType and operation name. The final parameter is an array of Objects that contain either actual data to be used as input, references to data stored at the proxy which are to be used input, or a combination of both. The array is transferred via SOAP and is mapped to an `ArrayOf_xsd_anyType`. Parameters must be in order and any input included in the array must be standard JAX-RPC supported types; the proxy will check this at runtime and exceptions will be thrown accordingly. References to data are represented as a time-stamped UID of type String. If the proxy receives any String elements in the input array it first checks to see if it represents a UID, if it does, the proxy considers it to be a reference to data stored locally, retrieves it and uses the actual data the reference points to as input to the Web service invocation. When this method is called the results from the Web service invocation are tagged with a UID and stored in the proxy's Hashtable, this UID is returned to the application as a reference to the actual data.

• **deliver** sets up data movement between proxies, moving it closer to the source of a Web service invocation. The first input parameter is a String containing the location of the recipient proxy. Each element in the second input parameter, `String[]` represents one reference to a blob of data stored at the proxy. Once invoked by an application the proxy will retrieve all the data the references point to, insert it into a hashtable and invoke the `stage` method on the recipient proxy; currently data is moved using SOAP, however we are exploring the use of protocols such as GridFTP for large data transfer. An acknowledgement is returned represented as a `boolean`.

• **stage** is used to actually transfer a set of data from one proxy to another. This method is called from within the `deliver` method on the recipient proxy and moves the data (packaged as a Hashtable) to the recipient proxy. An acknowledgement is returned, represented as a `boolean`.

• **returnData** can be used to retrieve stored data from a proxy when it is needed on a user's desktop, e.g. to obtain the final results at the end of a workflow. Once invoked, the proxy iterates the input array (`String[]`), which contains references to data, storing them in an array of type Object. This array is then returned to the invoking application.

• **flushTempData** is a house keeping method and is called to remove data from a proxy which is no longer required for any workflow components. This method takes a list of references to data, `String[]` and returns a `boolean` if successful.

• **addService/removeService** is used to instruct a proxy to maintain a new Web service, adding the WSDL to its repository or remove it from a proxy's control. The input String represents the WSDL of that new service.

• **listOperations** given a WSDL and a port type this method returns a `String[]` where each element is the name of an operation.

• **listOpParameters** given a string containing a WSDL which the proxy maintains, the port type and the name of an operation this method returns a `String[]` containing the types expected as input to an operation.

• **listOpReturnType** returns the return type information (represented as a String) of an operation given a WSDL, port type and operation name.

• **listServices** is used to query a proxy about which Web services it is currently maintaining. This information is returned in a `String[]`; each element represents one WSDL.

Proxies throw the following set of exceptions:

• **InvocationParameterError** is thrown if the service details (used an input) are not maintained by a proxy or if the types and/or number of parameters used in an input array do not match the actual Web service interface that the proxy is to invoke.

• **VariableNotFoundError** is thrown if there are any references to a WSDL or data which cannot be found at a proxy.

• **ServiceInvocationError** will be thrown if there are any faults with the actual Web service invocation, e.g. network failure, time-out etc.

• **ProxyAdminError** is thrown if an application is trying to add a Web service which is already maintained by the proxy, or if the WSDL location is invalid.

```
public interface proxy {
   //Proxy CORE methods
   public String invoke(String wsdl, String port, String op_name, Object[] params)
           throws InvocationParameterError, VariableNotFoundError, ServiceInvocationError;
   public boolean deliver(String proxy_wsdl, String[] dataToMove)
           throws VariableNotFoundError, ServiceInvocationError;
   public boolean stage(Hashtable dataToMove)
           throws ServiceInvocationError;
   public Object[] returnData(String[] dataToReturn)
           throws VariableNotFoundError;
   public boolean flushTempData(String [] dataToRemove)
           throws VariableNotFoundError;

   //Proxy ADMIN methods
   public void addService(String wsdl)
           throws ProxyAdminError;
   public void removeService(String wsdl)
           throws VariableNotFoundError;
   public String[] listOperations(String wsdl, String port)
           throws VariableNotFoundError;
   public String[] listOpParameters(String wsdl, String port, String op_name)
           throws VariableNotFoundError;
   public String[] listOpReturnType(String wsdl, String port, String op_name)
           throws VariableNotFoundError;
   public String[] listServices();
}
```

Fig. 3. Proxy API

D. Application to the Example Scenario

Referring back to our earlier scenario, Figure 4 represents a solution using our hybrid architecture. Active components in the scenario are coloured grey. The workflow has been described using a standard workflow language (e.g. BPEL) and is enacted by a centralised orchestration engine. It is important to note that the choice of workflow language is entirely based on the user's preference and doesn't affect the proxy architecture. The workflow explicitly interacts with the proxy when necessary. In order to orchestrate the workflow the following process takes place:

• **Phases 1-2:** The first step in the workflow execution involves making an invocation to `WS-1`, however instead of contacting the service directly, a call is made to a proxy (`P-1`) which has been installed on the same server as the Web service. A call to the `invoke` operation is made passing the name of the Web service, port type and operation to be invoked, along with any required input parameters. Once received, the proxy spawns a new thread of control and invokes the required operation, passing in the necessary input parameters. The output from the service invocation, in this case `R-WS1` is passed back to the proxy, tagged with a unique identifier (for reference later, e.g. retrieval, deletion etc.) and stored within the proxy; there is a requirement that the proxy has enough memory to store the results. Instead of the proxy directly passing the data back to the orchestration engine, a reference to the data, `$R-WS1` is returned. In a standard orchestration scenario the results of the Web service invocation would have first been moved to the orchestration engine and then moved to where they are needed at the analysis Web service. However, as the proxy has been installed on the same server as `WS-1`, the data can be transferred locally between the proxy and the Web service and did not have to move over a Wide Area network. The only thing returned to the orchestration engine was a reference to the output of the service invocation, `$R-WS1`. This process is repeated for `WS-2` and `WS-3`.

• **Phases 3-4:** The output from the Web service invocations are needed as input to the next service in the workflow, the analysis Web service. The orchestration engine contacts the peer storing the data, instructing it where to send the output from the last three Web service invocations. This is achieved by using the `deliver` command and passing as input the three references `$R-WS1, $R-WS2, $R-WS3` along with the WSDL address of the peer which is sitting in-front of the analysis Web service. Once this invocation is received by `P-1`, the proxy retrieves the stored data and transfers it across the network by invoking the `stage` operation on `P-A`. The data is then stored at `P-A` and if successful an acknowledgement message is sent back to `P-1` which is returned to the orchestration engine.

• **Phases 5-6:** The next stage in the workflow requires using the output from the first three services as input to the analysis Web service. In order to achieve this the orchestration engine passes the name of the service, port type, operation to invoke and the references to the output, which is required as

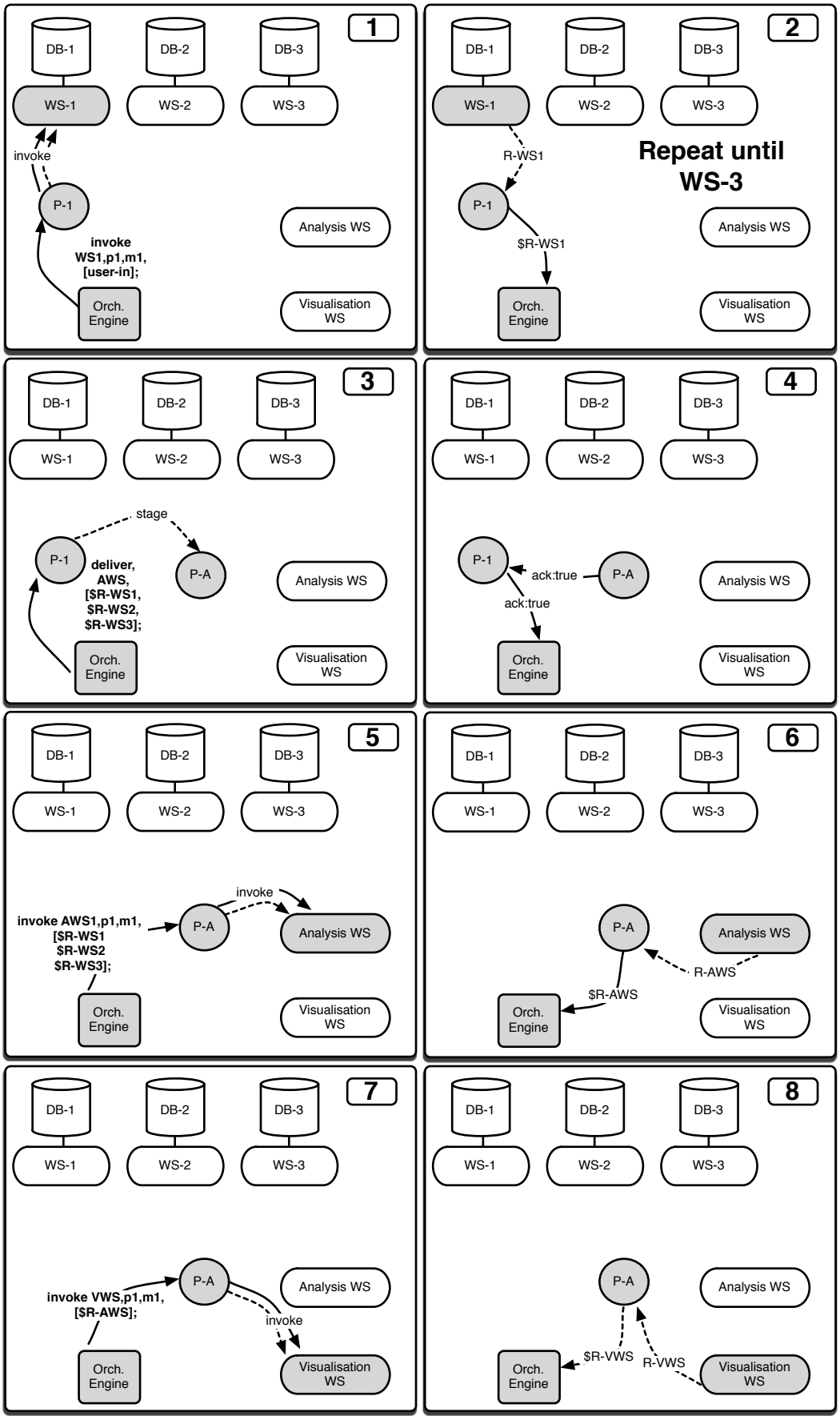

Fig. 4. Proxy architecture applied to the example scenario

input, in this case `$R-WS1, $R-WS2, $R-WS3`. The proxy then moves the data across the local network and invokes the operation which has been specified as input. The output, `R-AWS` is again stored locally on the proxy and a reference `$R-AWS` is passed back to the orchestration engine.

• **Phases 7-8:** Finally, the output from the analysis Web service is required as input to the visualisation Web service. In this case both services are maintained by the same proxy. The orchestration engine contacts the proxy, passing the name of the service, port type, operation to be invoked and the reference to the data, `$R-AWS`. The proxy invokes the required operation on the visualisation Web service and the results are transferred over the local network and stored at the proxy. A reference to the data, `$V-AWS` is returned to the orchestration engine; the actual data can be retrieved at any time from the proxy using the `returnData` operation.

E. Fault Handling

Fault handling is more complex when applications are built from a number of distributed components. Unlike a pure choreography model, our architecture maintains centralised control making it easier to detect and handle failures. Faults occur on four levels: *workflow specification faults*, when the workflow used as input to the orchestration engine is incorrect, *proxy faults*, when a proxy is down or overloaded, *Web service faults*, when a proxy is working but the Web service it is invoking on a user's behalf fails and *network faults*.

As an example consider the following Web service failure from Phase 1 of our scenario. The orchestration engine wants to invoke `WS-1` through the proxy, so a request is made to `P-1` which forwards the request. However when `P-1` makes an invocation to `WS-1` a network fault occurs telling the proxy that the Web service cannot be reached. As centralised control is maintained this fault can be relayed directly to the orchestration engine. If this scenario has been anticipated in advance and failure concerns have been built into the workflow specification then the orchestration can execute those alternatives, e.g. try the same request at a different Web service containing the same data. With a pure choreography model this is more difficult as failure would have to be modelled using time-outs between the sending and receiving processes.

A similar process would unfold for a proxy failure. Consider another example referring to Phase 3 of our scenario. Proxy `P-1` wants to deliver the data set `$R-WS1, $R-WS2, $R-WS3` to proxy `P-A`, however when `P-1` invokes `P-A` it discovers the proxy is unreachable and the data cannot be delivered. This failure gets relayed back to the orchestration engine which can take compensation action, e.g. try another proxy which is near the analysis Web service, or make a standard invocation to the Web service directly.

III. Related Work

There are a limited number of research papers which have identified the problem of a centralised approach to Web services orchestration when dealing with data-centric workflows. This Section will outline the main approaches which sit between standard orchestration and choreography techniques.

A. Flow-based Infrastructure for Composing Autonomous Services (FICAS)

The Flow-based Infrastructure for Composing Autonomous Services or FICAS [8] is a distributed data-flow architecture for composing software services into what the authors label *mega-structures* or workflow as it's more commonly known. Composition of the services in the FICAS architecture is specified using the *Compositional Language for Autonomous Services (CLAS)*, which is essentially a sequential specification of the relationships among collaborating services. This CLAS program is then translated by the build-time environment into a a control sequence that can be executed by the FICAS runtime environment. A centralised coordinator is in charge of sending and receiving all of the control messages which are the result of the workflow execution. Software applications are wrapped by an autonomous service mediator which facilitates data flow messages to pass directly between services without having to be transferred through a centralised server. FICAS has defined the autonomous service access protocol (ASAP), a protocol for accessing the services which uses asynchronous non-blocking communication.

Although FICAS is an architecture for decentralised orchestration it does not deal directly with modern standards and is a prototype and proof of concept. The issue of Web services integration is not addressed, nor does it discuss how this architecture could be incorporated into an orchestration language such as the de-facto standard, BPEL. More importantly FICAS is intrusive to the application code as each application that is to be deployed needs to be wrapped with a FICAS interface. In contrast, our proxy approach is more flexible as the services themselves require no alteration and do not even need to know that they are interacting with a proxy. Furthermore our proxy approach introduces the concept of passing references to data around and deals directly with modern workflow standards.

B. Service Invocation Triggers

Service Invocation Triggers [1], or simply triggers are also a response to the problem of centralised orchestration engines when dealing with large-scale data sets. The concept is based on the installation of a proxy which sits between the service which needs to be invoked and a client application. Triggers collect the required input data before they invoke a service, forwarding the results directly to where the data is required, this avoids the problem of passing it through a centralised orchestration engine. For this decentralised execution to take place, a workflow (expressed in a standard language, e.g. BPEL) must be deconstructed into sequential fragments which contain neither loops nor conditionals and the data dependancies must be encoded within the triggers themselves. For triggers to be effective they must be installed to the service as close as possible on a 1:1 basis.

The approach outlined by our paper and Service Invocation Triggers both rely on proxies to solve the problem of

decentralised orchestration when dealing with large data-sets. While Triggers address the issue of decentralised control, to realise these benefits their architecture is based around a pure choreography model, which as discussed throughout this paper has many extra problems associated with it. Furthermore, before execution can begin the input workflow must be deconstructed into sequential fragments, these fragments cannot contain loops and must be installed at a trigger; this is a rigid and limiting solution and is a barrier to entry for the use of proxy technology. In contrast with our proxy approach, because data references are passed around, nothing in the workflow has to be deconstructed or altered, which means standard orchestration languages such as BPEL can be used to coordinate the proxies. Finally, Triggers does not deal with modern Web service standards.

C. Other Relevant Techniques in Data Flow Optimisation

• OGSA-DAI [5] is a middleware product which supports the exposure of data resources, such as relational or XML databases, on to Grids. This middleware facilitates data streaming between local OGSA-DAI instances.

• Graph-forwarding is a technique [3] applied to distributed Objects, allowing the results of an RPC to be forwarded to the next object to invoke instead of the invoking object. This technique is similar in nature but doesn't address the issues concerning service composition through workflow technology.

IV. Conclusions and future work

This paper has introduced a hybrid approach to workflow enactment in the context of Web services, based on centralised control flow, distributed data flow. Our approach involves deploying proxies in the vicinity of Web services, which then allow for a more efficient data flow between workflow steps than possible in an orchestration model. As control flow is still centralised, this model has an advantage over the choreography model in fault detection and error handling. We summarise the main advantages of our hybrid model as:

• **Reduction in data transfer:** Unlike the standard orchestration model, proxies can exchange data flow messages directly with one another avoiding the need to pass large quantities of intermediate data through a centralised server. This reduces the amount of data that is transferred, just how much is currently being evaluated on the Planet Lab [12] framework.

• **Transition is non-disruptive:** The architecture can be deployed without disrupting current services and with minimal changes in the workflows that make use of them.

• **Flexible model:** The hybrid model and the architecture we have introduced allows mixing vanilla Web services with Web services served through a proxy. This flexibility allows a gradual change of infrastructures, where one could concentrate first on improving data transfers between services that handle large amounts data.

• **Simplicity of deployment:** The proxy services can be installed without the need for writing any additional code. Configuration can be done remotely and dynamically. It simply requires the whereabouts of WSDL descriptions for any services that will be enabled through the proxy.

• **Non-intrusive deployment:** A proxy need not be installed on the same server as the Web service, and does not interfere with the current vanilla Web service as is the case with pure choreography models. e.g. WS-CDL. However, to gain more performance, the proxy should be as close as possible to the Web services it is enabling.

The main limitations of the current architecture are:

• **Security:** Scenarios can be envisaged where malicious users overload a proxy or add and remove Web services from a proxy's control. Although our system is only a prototype, to allow live deployment, security issues will have to be addressed.

• **Time-out of cached data:** Proxies keep a cached copy of results from the vanilla Web service until data is successfully transferred to all the next steps in the workflow. When large amounts of data are concerned this may introduce problems as the data will have to be properly maintained by deleting redundant data.

We intend to create a mechanism which automatically transforms workflows that adhere to an orchestration model (e.g. BPEL) to our hybrid model, removing the need to manually call proxy functions. To increase the efficiency of data transfer, we are integrating several data transportation mechanisms into the proxies, e.g. GridFTP. Performance analysis is currently being conducted on Planet Lab and our architecture is being applied to live workflows in the e-Science community.

References

[1] Walter Binder, Ion Constantinescu, and Boi Faltings. Decentralized Ochestration of Composite Web Services. In *Proccedings of the International Conference on Web Services, ICWS'06*, pages 869–876. IEEE Computer Society, 2006.

[2] E. Deelman and et al. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[3] Andrew. S. Grimshaw, Jon. B. Weissman, and W.T. Strayer. *Portable Run-time Support for Dynamic Object-Oriented Parallel Processing*, volume 14(2) of *Transactions on Computer Systems*. ACM, May 1996.

[4] David Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, January 1995.

[5] K. Karasavvas and et al. Introduction to OGSA-DAI Services. In *Lecture Notes in Computer Science*, volume 3458, pages 1–12, May 2005.

[6] N. Kavantzas and et al. Web Services Choreography Description Language (WS-CDL) Version 1.0, November 2005.

[7] David Liu, Kincho H. Law, and Gio Wiederhold. Analysis of Integration Models of Service Composition. In *Proceedings of Third International Workshop on Software and Performance*, pages 158–165. ACM Press, 2002.

[8] David Liu, Kincho H. Law, and Gio Wiederhold. Data-flow Distribution in FICAS Service Composition Infrastructure. In *Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems*, 2002.

[9] A.R. Thakar, A.S. Szalay, and J. Gray. From FITS to SQL - Loading and Publishing the SDSS Data. In *Astronomical Data Analysis Software and Systems XIII*, volume 314, 2003.

[10] The OASIS Committee. Web Services Business Process Execution Language (WS-BPEL) Version 2.0, April 2007.

[11] Apache Web Services Project (AXIS): http://ws.apache.org/axis.

[12] Planet Lab: http://www.planet-lab.org/.

[13] Jun Zhao and et al. The Origin and History of in-silico Experiments. In *Proceedings of the UK e-Science all hands meeting*, September 2004.