DECENTRALISED ORCHESTRATION OF SERVICE-ORIENTED SCIENTIFIC WORKFLOWS

Adam Barker
School of Computer Science, University of St Andrews, St Andrews, U.K.

Rajkumar Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory
Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia

Keywords: Service-oriented architectures, Optimisation, Scientific workflows, Data deluge.

Abstract: Service-oriented workflows in the scientific domain are commonly composed as Directed Acyclic Graphs (DAGs), formed from a collection of vertices and directed edges. When orchestrating service-oriented DAGs, intermediate data are typically routed through a single centralised engine, which results in unnecessary data transfer, increasing the execution time of a workflow and causing the engine to become a performance bottleneck. This paper introduces an architecture for deploying and executing a service-oriented DAG-based workflows across a peer-to-peer proxy network. A workflow is divided into a set of vertices, disseminated to a group of proxies and executed without centralised control over a peer-to-peer proxy network. Through a Web services implementation, we demonstrate across PlanetLab that by reducing intermediate data transfer and by sharing the workload between distributed proxies, end-to-end workflows are sped up. Furthermore, our architecture is non-intrusive: Web services owned and maintained by different institutions do not have to be altered prior to execution.

1 INTRODUCTION

Service-oriented architectures are an architectural paradigm for building software applications from a number of loosely coupled distributed services. This paradigm has seen wide spread adoption through the Web services approach, which has a suite of core standards (e.g., XML, WSDL and SOAP) to facilitate interoperability.

These core standards however do not provide the rich behavioural detail necessary to describe the role an individual service plays as part of a larger, more complex collaboration. Co-ordination of services is often achieved through the use of workflow technologies. As defined by the 'Workflow Management Coalition' (Hollingsworth, 1995), 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, either human or machine) to another for action, according to a set of procedural rules.

Workflows in the scientific community (Barker and van Hemert, 2008) are commonly modelled as Directed Acyclic Graphs (DAGs), formed from a collection of vertices (units of computation) and directed edges. The Genome Analysis and Database Update system (GADU) (Sulakhe et al., 2008), the Southern California Earthquake Centre (SCEC) (Deelman and et al., 2006) CyberShake project, and the Laser Interferometer Gravitational-Wave Observatory (LIGO) (Taylor et al., 2006) are examples of High Performance Computing applications composed using DAGs. DAGs present a *dataflow view*, here data is the primarily concern, workflows are constructed from processing vertices and data transport edges.

Taverna (Oinn and et al, 2004) is an example of a popular graphical Web service composition tool used primarily in the life sciences community in which workflows are represented as DAGs. Graph vertices can be one of a set of service types: WSDL Web services, BeanShell (lightweight scripting for Java) components, String constants etc. Services are given input and output ports which correspond to individual input and output variables. Edges are then formed by connecting services together by mapping output ports with input ports.

Barker A. and Buyya R..
DECENTRALISED ORCHESTRATION OF SERVICE-ORIENTED SCIENTIFIC WORKFLOWS.
DOI: 10.5220/0003384302220231
In *Proceedings of the 1st International Conference on Cloud Computing and Services Science* (CLOSER-2011), pages 222-231
ISBN: 978-989-8425-52-2

1.1 Motivating Scenario: Calculating Redshift

At this point, in order to put our motivation and problem statement into perspective, it is useful to consider an illustrative scenario. The Redshift scenario is taken from the AstroGrid (Allan et al., 2010) (UK e-Research project) science use-cases and involves retrieving and analysing data from multiple distributed resources. This scenario is representative of a class of large-scale scientific workflows, where data and services are made available through a Web service. It will be referenced throughout the remainder of this paper. Figure 1 is a representation of the AstroGrid redshift scenario in Taverna.

Photometric Redshifts use broad-band photometry to measure the Redshifts of galaxies. While photometric Redshifts have larger uncertainties than spectroscopic Redshifts, they are the only way of determining the properties of large samples of galaxies. This scenario describes the process of querying a group of distributed databases containing astronomical images in different bandwidths, extracting objects of interest and calculating the relative Redshift of each object.

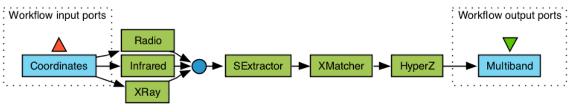

Figure 1: AstroGrid scenario – Taverna representation. Workflow inputs are the RA and DEC coordinates, services are represented as rectangles, links correspond to the flow of data between services.

The scenario represents a workflow and begins with a scientist inputting the RA (right ascension) and DEC (declination) coordinates into the system, which define an area of sky. These coordinates are used as input to three remote astronomical databases; no single database has a complete view of the data required by the scientist, as each database only stores images of a certain waveband. At each of the three databases the query is used to extract all images within the given coordinates which are returned to the scientist. The images are concatenated and sent to the SExtractor (Bertin and Arnouts,) tool for processing. SExtractor scans each image in turn and uses an algorithm to extract all objects of interest (positions of stars, galaxies etc.) and produces a table for each of the wavebands containing all the data. A cross matching tool is then used to scan all the images and produce one table containing data about all the objects of interest in the sky, in the five wavebands. This table is then used as input to the HyperZ[1] algorithm which computes the photometric Redshifts and appends it to each value of the table used as input. This final table consists of multi-band files containing the requested position as well as a table containing for each source all the output parameters from SExtrator and HyperZ, including positions, magnitudes, stellar classification and photometric Redshifts and confidence intervals; the final table is returned to the user.

1.2 Problem Statement

Although service-oriented workflows can be composed as DAGs using a dataflow model, in reality they are orchestrated from a single workflow engine, where intermediate data are typically routed through a centralised engine.

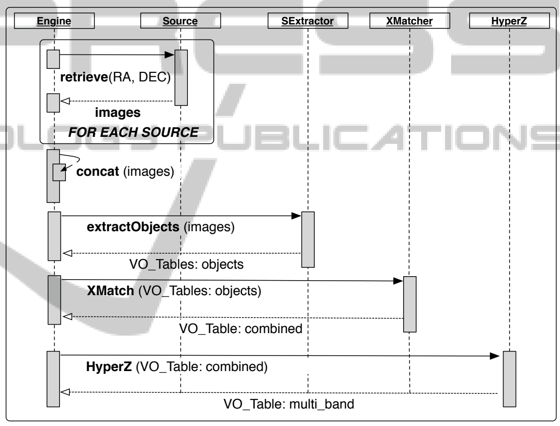

Figure 2: UML Sequence diagram: AstroGrid scenario.

Figure 2 is a UML Sequence diagram displaying how the AstroGrid workflow is orchestrated. The initial RA and DEC coordinates are used as input to each of the three source databases: Radio, Infrared and XRay. Each source database then returns a set of images to the workflow engine. These images are then combined and passed through the SExtractor, XMatcher and HyperZ services. Finally, the HyperZ service returns the Multiband table as output.

In the AstroGrid scenario, output from each of the source databases and processing services passes via the workflow engine, in order to be passed to the next service in the workflow chain. When one is orchestrating Web services from a tool such as Taverna, the workflow engine becomes a bottleneck to the performance of a workflow. Large sets of intermediate data which are consistent with scientific workflows are routed via the workflow engine, which results in unnecessary data transfer, increasing the execution

[1]http://webast.ast.obs-mip.fr/hyperz/

time of a workflow and causing the engine to become a bottleneck to the execution of an application.

1.3 Paper Contributions and Structure

This paper proposes a novel architecture for accelerating end-to-end workflows by deploying and executing a service-oriented workflow (composed as a DAG) across a peer-to-peer proxy network. Individual proxies are deployed at strategic network locations in order to exploit connectivity to remote Web services. By 'strategic' we could refer to one of many factors including network distance, security, policy etc. Proxies together form a proxy network which facilitate a number of functions such as service invocation, routing of intermediate data and data caching.

By breaking up a workflow and disseminating it across a peer-to-peer network, workflow bottlenecks associated with centralised orchestration are partially removed, intermediate data transfer is reduced, by sharing the workflow across distributed proxies and applications composed of interoperating Web services are sped up.

Importantly, our proposed architecture is a non-intrusive solution, Web services do not have to be redeployed or altered in any way prior to execution. Furthermore, a DAG-based workflow defined using a visual composition tool (e.g., Taverna) can be simply translated into our DAG-based workflow syntax and automatically deployed across a peer-to-peer proxy network.

A Java Web services implementation serves as the basis for performance analysis experiments conducted over PlanetLab (Chun et al., 2003). These experiments empirically demonstrate how our proposed architecture can speed up the execution time of a workflow when compared to standard orchestration. Although this paper focuses on Web services, the concept is generic and can be applied to other classes of application, i.e., High Performance Computing, Cloud Computing etc.

This paper is structured as follows: Section 2 introduces the architecture of an individual proxy and multiple proxies which together form a proxy network. A syntax is introduced for service-oriented DAG-based workflows and the algorithms which divide a workflow, assign and enact a workflow across a proxy network are described. A Java Web services implementation, which serves as a platform for performance analysis is also discussed. Section 3 describes the performance analysis experiments. Related work is discussed in Section 4. Finally conclusions and future work are addressed in Section 5.

2 PROXY ARCHITECTURE

Within our architecture a proxy is a middleware component that is deployed at a strategic location to a Web service or set of Web services. For the purposes of this paper, by strategic we mean in terms of network distance; as closely as possible to an enrolled service, i.e., on the same Web server or within the same domain, such that communication between a proxy and a Web service takes place over a local, not Wide Area Network. Proxies are considered to be volunteer nodes and can be arbitrarily sprinkled across a network, importantly not interfering with currently deployed infrastructure.

A proxy is generic and can concurrently execute any workflow definition. In order for this to be possible, the workflow definition is treated as an executable object which is passed between proxies in a proxy network. Proxies invoke Web services on behalf of the workflow engine, storing any intermediate data at the proxy. Proxies form peer-to-peer proxy networks and can route data directly to one another, avoiding the bottleneck problems associated with passing intermediate data through a single, centralised workflow engine.

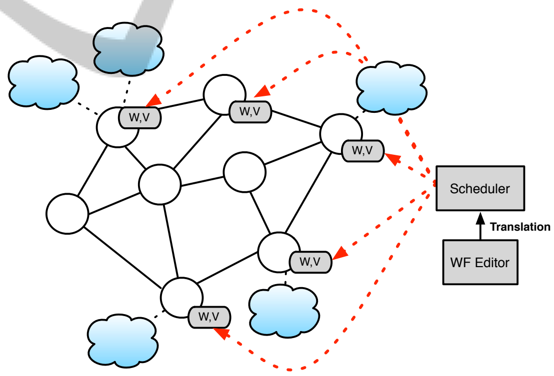

Figure 3: Proxy architecture: Web services represented by clouds, proxies by circles, the workflow definition and vertex (W,V) by a rounded rectangle.

Figure 3 shows a high level architectural diagram of a proxy network. A user designs a service-oriented DAG-based workflow using a visual workflow editor such as Taverna. A scheduling service assigns workflow vertices to proxies, the unique identifier of each proxy is then spliced into the workflow definition. Each proxy then is passed an entire copy of the workflow definition. Once a proxy receives the workflow definition, it executes its assigned set of vertices (once all dependencies are resolved) and passes any intermediate data according to the directed edge definition. Once deployed a DAG-based workflow is executed without any centralised control over a peer-to-peer proxy network.

The following subsections describe in detail how a user designs a workflow, how the workflow definition is divided and assigned to a set of proxies, deployed across a proxy network and enacted.

2.1 Workflow Definition

A workflow is specified as a DAG according to the syntax displayed in Figure 4. It is important to note that the DAG syntax is not a novel contribution of this paper, it is primarily a way to describe the algorithms and architecture of our proposed approach. We have taken inspiration from the Taverna SCUFL language (Oinn and et al, 2004), our syntax is a simplified version which does not support the additional processor types such as BeanShell etc. A workflow relies on static binding at deployment time, dynamic binding methods are left to future research.

Workflow	::=	ID_w, {Vertex}, {Edge}, ID_s
Vertex	::=	`vertex`(ID_v, Processor)
Processor	::=	WS | Input | Output
WS	::=	`s`($\mathrm{Config}^{(k)}$, {Inport}, {Outport}, ID_p)
Input	::=	`input`(value, Outport)
Output	::=	`output`(value, Inport)
Inport	::=	`in`(ID_{in}, Type, [digit])
Outport	::=	`out`(ID_{out}, Type)
ID	::=	ID_w | ID_s | ID_v | ID_p | ID_{in} | ID_{out}
Type	::=	XML RPC Types
Config	::=	$\langle$name, value$\rangle$
Edge	::=	ID_v:$\mathrm{ID}_{out} \rightarrow$ {ID_v:ID_{in}}

Figure 4: Workflow definition syntax.

A workflow is labelled with a unique identifier $\mathtt{ID}_w$ and consists of a set of vertices {`Vertex`} and a set of edges {`Edge`}. A vertex is given a globally unique identifier $\mathtt{ID}_v$ and can consist of one of a set of processor types. As this paper focuses on service-oriented workflows, processors are Web service definitions `WS` or `input` and `output` variables.

A Web service is defined firstly as a list of configuration pairs $\mathtt{Config}^{(k)}$ which are simply $\langle$name, value$\rangle$ pairs and define the information necessary to invoke an external Web service: WSDL location, operation name etc. Secondly, a set of input ports {`Inport`}; each inport is given a unique identifier within a processor $\mathtt{ID}_{in}$ and a Type definition, types map to the standard set of XML RPC Types[2]. The final parameter (optional) defines how many inputs are expected at a given inport. A set of output ports {`Outport`}; each outport is given a unique identifier within a processor $\mathtt{ID}_{out}$ and a `Type` definition. The final parameter of a processor is $\mathtt{ID}_p$, which represents the globally unique identifier (mapping to an individual IP address) of a proxy which is executing a given vertex; these are initially `null` and spliced in before the workflow definition is disseminated to a set of proxies. Proxy identifiers are included in the workflow definition so that individual proxies can communicate with one another when executing a workflow. $\mathtt{ID}_s$, the final parameter of a Workflow definition is the location (IP address) of the scheduling service, this is also spliced in before the workflow is disseminated so that the final output from a workflow can be passed back to the user.

[2]http://ws.apache.org/xmlrpc/types.html

The final processor types supported are `Input` (used as input to a Web service) and `Output` variables (used as output from a Web service). An Input variable is defined as a `value`, which is the actual value assigned to the variable and an outport definition. An Output variable is defined as a `value`, which is initially a wildcard and an inport definition. The Types supported are the same set of XML RPC types.

In order to complete the workflow, a set of directed edges are formed which constitute a dataflow mapping from processor inports to processor outports. This is specified by providing the following mapping: $\mathtt{ID}_v\mathtt{:ID}_{out} \rightarrow \{\mathtt{ID}_v\mathtt{:ID}_{in}\}$. The types of the outport to inport mappings must match and are enforced by the workflow editor.

2.2 Example Definition

Figure 5 is a representation of the AstroGrid scenario in the workflow definition syntax, Figure 6 is a corresponding diagrammatic representation. With reference to Figure 5, within the scope of the workflow identifier `calculate_redshift`, eight vertices are defined: `ra`, `dec` represent the workflow input parameters, the variables are defined by the physical values which are transferred via the outports `ra_output` and `dec_output`. As the workflow output needs to be written back to a user's desktop, the vertex `wf_o` represents the final workflow output which will eventually be written to the inport `multi_band`; this output will then be passed back to the scheduling service which initiated the workflow.

The remaining vertices are `WS` definitions, `radio`, `infra` and `xray` are the distributed data sources, containing data from each of the required wave lengths; `tools` represents the co-located services `SExtractor` and `XMatcher`; finally `z` is the `HyperZ` processing service. Each of the service definitions contains typed inport and outport definitions; note the `3` in the `tools`

```
calculate_redshift,

//RA, DEC and Output Vertices
{vertex(ra, input(100, {out(ra_output, String)})),
vertex(dec, input(50, {out(dec_output, String)})),
vertex(wf_o, output(_, {in(multi_band, Object[])})),

//Source Vertices
vertex(radio, s(config, {in(ra_input, String),
in(dec_input, String)}, {out(image_set, byte[])}, _)),
vertex(infra, s(config, {in(ra_input, String),
in(dec_input, String)}, {out(image_set, byte[])}, _)),
vertex(xray, s(config, {in(ra_input, String),
in(dec_input, String)}, {out(image_set, byte[])}, _)),

//Processing Vertices
vertex(tools, s(config, {in(images, byte[], 3)},
{out(combined, Object[])}, _)),
vertex(z, s(config, {in(combined, Object[])},
{out(multi_band, Object[])})), _},

//Edge definitions
{ra:ra_output -> radio:ra_input, infra:ra_input,
xray:ra_input,
dec:dec_output -> radio:dec_input, infra:dec_input,
xray:dec_input,
radio:image_set -> tools:images,
infra:image_set -> tools:images,
xray:image_set -> tools:images,
tools:combined -> z:combined,
z:multi_band -> wf_o:multi_band}, _
```

Figure 5: AstroGrid scenario workflow definition.

inport `images` states that 3 inputs (which will be merged) are required, for simplicity `config` represents the concrete details of individual Web services. The `wildcard` at the end of each service definition is the unique identifier of the proxy $\mathtt{ID}_p$ which is spliced in before the workflow definition is disseminated to a set of proxies. A set of `Edge` definitions connect `vertex outports` with `vertex inports` according to the flow of data in the AstroGrid scenario.

2.3 Web Services Implementation

The proxy architecture is available as an open-source toolkit implemented using a combination of Java (version 1.6) and Apache Axis (version 2) Web services (Apache Axis,), it consists of the following core components:

• **Registry Service.** When a proxy is deployed it is automatically enrolled with the registry service, which contains global knowledge of distributed proxies. The registry service logs data of previous successful interactions and proxies are polled to ensure availability.

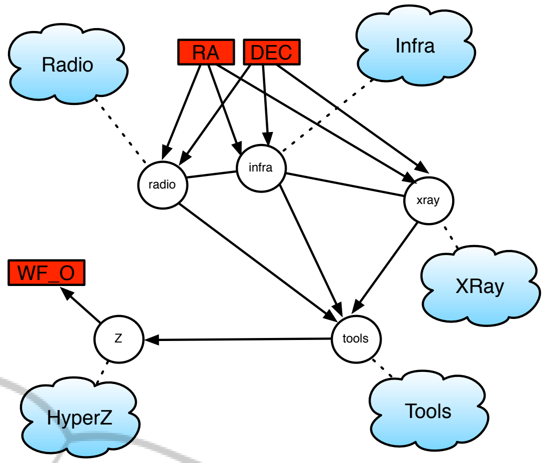

Figure 6: Possible proxy configuration for the AstroGrid scenario: Edges are directed and show dataflow between proxies. Workflow inputs (RA, DEC) and outputs (WF_O) are also labelled. For simplicity, tools represents the co-located services SExtractor and XMatcher. Workflow engine and scheduling service not shown.

• **XML Syntax.** The workflow syntax displayed in Figure 4 is encoded in XML, allowing the registry service to splice in the proxy identifiers and proxies. Type checking between outport and inport definitions is enforced at the syntax level.

• **Translation.** A translation component automatically converts workflows defined in the Taverna SCUFL dataflow language into our workflow specification syntax. Translations from other languages are possible, we have chosen Taverna SCUFL as it is a widely accepted platform, particularly in the life sciences community.

• **Scheduling Service.** Once a user has designed a workflow and it has been translated, the scheduling service (a local component) takes as input the workflow definition and consults the registry service, splicing in a unique proxy identifier for every vertex in a workflow definition. The scheduling service's IP address is spliced into the workflow definition, so that final output can be sent back to the user.

• **Proxy.** Proxies are made available through a standard WSDL interface, allowing them to be simply integrated into any standard workflow tool. As discussed further in Section 2.5, a proxy has two remote methods: `initiate` to instantiate a proxy with a workflow and a vertex, and `data`, allowing proxies to pass data to one another, which in our implementation is via SOAP messages over HTTP. Proxies are simple to install and configure and can be dropped into an AXIS container running on an application server, no specialised programming needs to take place in order to exploit the functionality.

2.4 Workflow Deployment and Vertex Assignment

For simplicity in the algorithm definition, we assume reliable message passing and service invocation, however, fault tolerance has been built into the corresponding Web services implementation. A proxy is generic and can concurrently execute any vertex from any workflow definition. In order for this to be possible, the workflow definition is treated as an executable object, which is passed between proxies in a proxy network. The workflow definition is passed to the scheduling service which needs to assign proxies to vertices. This process is formally defined by Algorithm 1.

Algorithm 1: Vertex assignment.

for each Vertex ID_v where $\mathrm{ID}_v \in \{\mathrm{Vertex}\}$ **do**
  if (Processor = WS) **then**
    $\mathrm{ID}_p \leftarrow$ `locate`(ID_v, WS)
    $\mathrm{WS.ID}_p \leftarrow \mathrm{ID}_p$
    $\{\langle \mathrm{ID}_v, \mathrm{ID}_p \rangle\} \leftarrow \{\langle \mathrm{ID}_v, \mathrm{ID}_p \rangle\} + \mathrm{ID}_v, \mathrm{ID}_p$
  end if
end for
for each Vertex ID_v where $\mathrm{ID}_v \in \{\langle \mathrm{proxy}, \mathrm{ID}_v \rangle\}$ **do**
  `initiate`(Workflow, ID_v)
end for

All proxies are enrolled with the registry service, which is a global directory of each proxy within the system. For each $\mathtt{ID}_v$ in $\{\mathtt{Vertex}\}$ a suitable proxy must be located, if the processor type is a service definition, i.e., not an input or output variable. In our existing implementation, the registry service selects 'suitable' proxies which are deployed with the same network domain as the `WS` it will eventually invoke. However, we are investigating optimisation techniques which will be addressed by further work, discussed in more detail in Section 5.

This suitability matching is performed by the scheduling service which in turn consults with the registry service. The scheduling service invokes the `locate` method on the registry service, which takes as input a vertex $\mathtt{ID}_v$ and a `WS` definition and returns the unique identifier of a proxy which will enact a given vertex $\mathtt{ID}_v$. This identifier is then spliced into the processor definition; before the assignment process begins all $\mathtt{ID}_p$ definitions are wildcards, each vertex (multiple vertices can be assigned to the same proxy) is then assigned before the workflow is disseminated. The proxy identifier is added so that proxies can communicate throughout the system globally.

The proxy identifier along with the vertex identifier are added to a set. Once the proxy assignment process is complete, the workflow definition and vertex a proxy is to assume $\mathtt{ID}_v$ is sent to each proxy in the set $\{\langle \mathtt{ID}_v, \mathtt{proxy} \rangle\}$. The remote method `initiate` is invoked on each proxy.

2.5 Workflow Execution

A proxy can concurrently execute any vertex from any workflow. With reference to Algorithm 2, in order to initiate a workflow, the remote method `initiate` is invoked on each proxy, given a workflow definition and $\mathtt{ID}_v$. The vertex definition $\mathtt{ID}_v$ is extracted from the workflow. If the vertex relies on data from other vertices, it must wait until all inports have been resolved. Therefore, each inport $\mathtt{ID}_{in}$ in $\{\mathtt{Inport}\}$ must be resolved before execution of $\mathtt{ID}_v$ can begin. This is achieved through the internal method `resolve_in` which checks if data for a given inport has been received; if the inport vertex definition is simply an `input` variable then the corresponding value is retrieved.

Algorithm 2: Vertex enactment.

`initiate`(Workflow, ID_v)
for each Inport ID_{in} where $\mathrm{ID}_{in} \in \{\mathrm{Inport}\}$ **do**
  `resolve_in`(ID_{in})
end for
$\{\mathrm{input}\} \leftarrow$ `get_input`(ID_w, ID_v)
results $\leftarrow$ `invoke`($\{\mathrm{config}\}$, $\{\mathrm{input}\}$)
for each Outport ID_{out} where $\mathrm{ID}_{out} \in \{\mathrm{Outport}\}$ **do**
  $\{\mathrm{ID}_v{:}\mathrm{ID}_{in}\} \leftarrow$ `resolve_out`($\mathrm{ID}_v{:}\mathrm{ID}_{out}$, $\{\mathrm{Edge}\}$)
  for each Vertex ID_v where $\mathrm{ID}_v \in \{\mathrm{ID}_v{:}\mathrm{ID}_{in}\}$ **do**
    if (Processor = WS) **then**
      $\mathrm{ID}_p \leftarrow \mathrm{WS.ID}_p$
      `data`(ID_w, ID_{in}, results)
      `delete`(results)
    else if (Processor = Output) **then**
      value $\leftarrow$ results
      `data`(ID_w, ID_s, results)
    end if
  end for
end for

Once all inport dependencies have been resolved, given the unique workflow identifier $\mathtt{ID}_w$ and $\mathtt{ID}_v$, the input data set $\{\mathtt{input}\}$ is retrieved through the internal method `get_input`. The proxy takes the WSDL location, operation name and other parameters defined in $\{\mathtt{config}\}$ and dynamically invokes the service using $\{\mathtt{input}\}$ as input to the Web service. Results are then stored locally at the proxy.

In order to determine where (i.e., which proxy) to send the output of a given service invocation, the $\{\mathtt{Edge}\}$ set is inspected which contains mappings from a vertex outport to a set of vertex inports. The set of inports which map to a corresponding outport is returned by the internal `resolve_out` method. In order to determine which proxy to send these data to, each vertex in this set is traversed and the location of the proxy, $\texttt{ID}_p$ is retrieved from the workflow definition.

The remote method `data` is invoked on the proxy $\texttt{ID}_p$, using the workflow identifier $\texttt{ID}_w$, the inport identifier $\texttt{ID}_{in}$ and the result data as input. Once received (confirmed by an acknowledgement) by the recipient proxy, these data are stored according to $\texttt{ID}_w$ and $\texttt{ID}_{in}$ and deleted from the sender proxy. If the outport corresponds to a `output`, this variable is written back to the scheduling service $\texttt{ID}_s$, which is running on a user's desktop. This process is repeated for each outport.

2.6 End-to-end Example

Figure 7 is a UML Sequence diagram which demonstrates the interaction between the scheduling service, registry service and the set of proxies in the AstroGrid scenario.

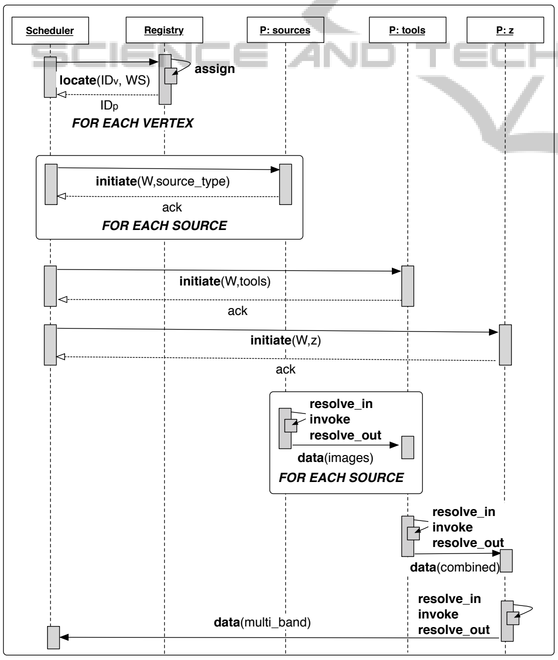

Figure 7: UML Sequence diagram: AstroGrid scenario proxy network.

3 PERFORMANCE EVALUATION

In order to validate the hypothesis that our architecture can reduce intermediate data transfer and speed up the execution time of a workflow, a set of performance analysis experiments have been conducted. Our architecture has been evaluated across Internet-scale networks on the PlanetLab framework. PlanetLab is a highly configurable global research network of distributed servers that supports the development of new network services.

The AstroGrid scenario described throughout this paper serves as the basis for our performance analysis. This scenario is representative (in terms of data size and topology) of a class of large-scale scientific workflows and has been configured as follows:

• **PlanetLab Deployment.** Data sources are a Web service which take as input an integer representing how much data the service is to return; the service then returns a Byte array of the size indicated in the input parameter. Analysis services are simulated via a sleep and return a set of data representative of the given input size. These data sources and analysis services were deployed over the PlanetLab framework.

• **Workflow Engine.** In order to benchmark our architecture two configurations of the AstroGrid scenario were set up: the first was executed on the completely centralised Taverna workflow (version 1.7.2) environment, the second was the same representation executed across a peer-to-peer proxy network according to the implementation described in Section 2.3.

• **Speedup.** The mean speedup is calculated by dividing the mean time taken to execute the workflow using standard orchestration (i.e., non-proxy, fully centralised) and dividing it by the mean time taken to execute the workflow using our proxy architecture, e.g., a result of 1.5 means that the proxy architecture executes 50% faster than standard orchestration.

• **Proxy Configurations.** Three different proxy configurations are shown: "same machine", here a proxy is installed on the same physical machine as the Web service it is invoking, "same domain", the proxy is installed on a different machine within the same network domain, and "distributed" where a proxy is installed on a node within the same country as the Web service it is invoking. In each configuration one proxy is responsible for one service.

• **Graphs.** Each configuration was executed 50 times across the PlanetLab framework. In each graph, the *y*-axis displays the mean speedup ratio (along with 95% confidence intervals from each of the 50 runs per configuration) and the *x*-axis displays the total volume of data flowing through the workflow. The number of services involved is independent of the mean speedup ratio as we have taken the mean ratio across a set of scaling experiments: we have scaled the initial data sources from 2 to 20 and repeated this while executing the AstroGrid DAG in reverse order. To prevent the data processing from influencing our evaluation, it has not been accounted for in the performance analysis experiments.

3.1 Experiment 1

Each of the data sources, analysis services and workflow engine were installed on separate PlanetLab nodes in the USA. As one can see from Figure 8 in all configurations our architecture outperforms the centralised workflow configuration. If one calculates the average across all data points for each of the experiments, the "same machine" configuration observes a speedup of 75%, the "same domain" configuration 49% and the "distributed" configuration 30%.

As the results demonstrate, the speedup is greatest when a proxy is deployed as closely as possible to the back-end Web service, i.e., on the same machine. The cost of the proxy to service data movement increases as the proxy moves further away from the service it is invoking, in the "same machine" configuration, the input and output of a service invocation is flowing over a LAN. However, in the most extreme case, the "distributed" configuration an average speedup of 30% is observed over all runs.

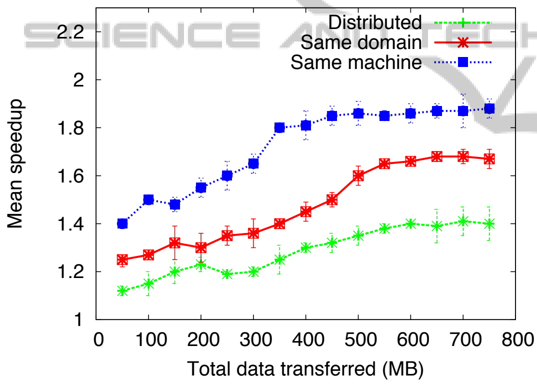

Figure 8: Experiment 1 – Mean speedup.

3.2 Experiment 2

In this configuration, each of the data sources and analysis services were deployed on separate PlanetLab nodes across the USA. However, the workflow engine was now even further distributed from the services, running from a desktop machine in Melbourne. As one can see from Figure 9, as the cost (network distance) increases from the workflow engine to the workflow services, the hop any intermediate data has to make increases in cost. As the cost of intermediate data transport increases, the benefit of using our architecture grows as intermediate data transport is optimised. To quantify, this speedup ranged from 68% to 153% speedup for the "same domain" configuration and 51% to 125% for the "distributed" configuration.

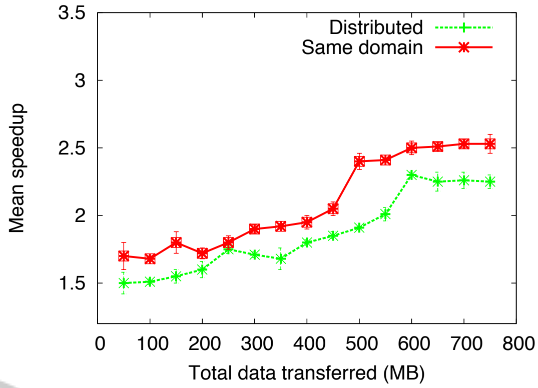

Figure 9: Experiment 2 – Mean speedup.

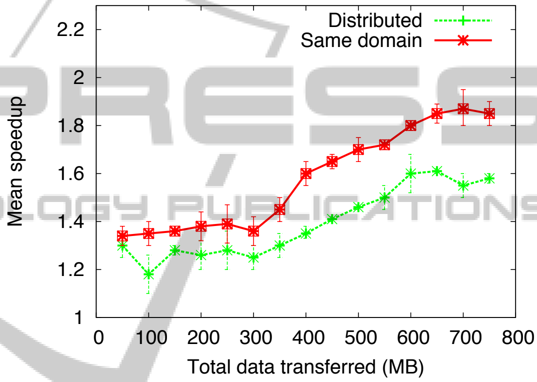

Figure 10: Experiment 3 – Mean speedup.

3.3 Experiment 3

In order to distribute the services further, the data sources were deployed on PlanetLab nodes in the USA, the analysis services deployed on nodes in Europe (France and Germany) and the workflow engine was running from a desktop machine in Melbourne. With reference to Figure 10, speedup ranged from 34% to 85% for the "same domain" configuration and 30% to 58% for the "distributed" configuration. In this experiment one can observe an increased cost in distributing the workflow definition to each of the proxies prior to enactment, demonstrated by the lack of increase in mean speedup at lower data sizes.

4 RELATED WORK

4.1 Techniques in Data Transfer Optimisation

In (Martin et al., 2008) the scalability argument made in this paper is also identified. The authors propose a methodology for transforming the orchestration logic in BPEL into a set of individual activities that coordinate themselves by passing tokens over shared, distributed tuple spaces. The model suitable for execution is called Executable Workow Networks (EWFN), a Petri nets dialect.

The concept of pointers in service-oriented architectures (Wieland et al., 2009) allows Web services to pass data by reference rather than by value. This has the advantage that the workow engine is disburdened of handling all data passing between the orchestrated Web Services, which helps to reduce network trafc and processing time.

Service Invocation Triggers (Binder et al., 2006) is an architecture for decentralised execution. Using the Triggers architecture, 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 nothing in the workflow has to be altered prior to enactment.

The *Flow-based Infrastructure for Composing Autonomous Services* or FICAS (Liu et al., 2002) is a distributed data-flow architecture for composing software services. FICAS is intrusive to the application code as each application that is to be deployed needs to be wrapped with a FICAS interface.

In (Chafle et al., 2004), an architecture for *decentralised orchestration* of composite Web services defined in BPEL is proposed. Our research deals with a set of challenges not addressed by this architecture: how to optimise service-oriented DAG-based workflows, how to automatically deploy a workflow across a set of volunteer proxy nodes given a workflow topology, where to place proxies in relation to Web services, how these concepts operate across Internet-scale networks.

In previous work (Barker et al., 2008b) (Barker et al., 2008a) we proposed Circulate, a proxy-based architecture based on a centralised control flow, distributed data flow model. This paper has focused on executing DAG-based workflows without centralised control and explored a richer set of proxy functionality.

4.2 Third-party Data Transfers

This paper focuses primarily on optimising service-oriented workflows, where services are: not equipped to handle third-party transfers, owned and maintained by different organisations, and cannot be altered in anyway prior to enactment. For completeness it is important to discuss engines that support third-party transfers between nodes in task-based workflows.

Directed Acyclic Graph Manager (DAGMan) (Condor Team,) submits jobs represented as a DAG to a Condor pool of resources. Intermediate data are not transferred via a workflow engine, instead they are passed directly from vertex to vertex. DAGMan removes the workflow bottleneck as data are transferred directly between vertices in a the DAG. *Triana* (Taylor et al., 2003) is an open-source problem solving environment. It is designed to define, process, analyse, manage, execute and monitor workflows. Triana can distribute sections of a workflow to remote machines through a connected peer-to-peer network.

5 CONCLUSIONS AND FURTHER WORK

Through a motivating scenario, this paper has introduced an architecture for deploying and executing a service-oriented workflow (composed as a DAG) across a peer-to-peer proxy network. This architecture partially avoids workflow bottlenecks associated with centralised orchestration by: sharing the workload across distributed proxies, and reducing intermediate data transfer between interoperating services in a workflow. Importantly our proposed architecture is non-intrusive, Web services do not have to be altered in anyway prior to execution. Furthermore, users can continue to work with popular service-oriented DAG-based composition tools; our architecture translates DAG-based workflows (we have used Taverna SCUFL) into our workflow specification syntax, vertices are assigned, disseminated and enacted by an appropriate set of proxies.

A Web services implementation was introduced which formed the basis of our performance analysis experiments conducted over the PlanetLab framework. Performance analysis demonstrated across various network configurations that by reducing intermediate data transfer end-to-end workflows are sped up, in the best case from 68% to 192%.

Further work includes the following:

• **Expression of Workflows.** This paper has focused on DAG-based workflows. Further work will address aligning the architecture with business process notations.

• **Peer-to-peer Registry.** The registry service is currently centralised. Peer-to-peer techniques utilising Chord (Stoica et al., 2001) are being investigated, with the view to improving scalability.

ACKNOWLEDGEMENTS

This work is supported by an Engineering and Physical Sciences Research Council (EPSRC) / Joint Information Systems Committee (JISC) grant (EP/I034327/1) and an Australian Research Council (ARC) Linkage Project grant.

REFERENCES

Allan, P., Bentley, B., and et al. (2001 [26/02/2010]). AstroGrid. Technical report, Available at: www.astrogrid.org.

Apache Axis. http://ws.apache.org/axis [22/02/2011].

Barker, A. and van Hemert, J. (2008). Scientific Workflow: A Survey and Research Directions. In Wyrzykowski, R. and et al., editors, *Seventh International Conference on Parallel Processing and Applied Mathematics, Revised Selected Papers*, volume 4967 of *LNCS*, pages 746–753. Springer.

Barker, A., Weissman, J. B., and van Hemert, J. (2008a). Eliminating the Middle Man: Peer-to-Peer Dataflow. In *HPDC '08: Proceedings of the 17th International Symposium on High Performance Distributed Computing*, pages 55–64. ACM.

Barker, A., Weissman, J. B., and van Hemert, J. (2008b). Orchestrating Data-Centric Workflows. In *The 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 210–217. IEEE Computer Society.

Bertin, E. and Arnouts, S. Sextractor: Software for source extraction, Astronomy and Astrophysics, Suppl. Ser., 117:393–404, 1996.

Binder, W., Constantinescu, I., and Faltings, B. (2006). Decentralized Ochestration of Composite Web Services. In *Proceedings of ICWS'06*, pages 869–876. IEEE Computer Society.

Chafle, G. B., Chandra, S., Mann, V., and Nanda, M. G. (2004). Decentralized orchestration of composite web services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143. ACM.

Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. (2003). Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12.

Condor Team. www.cs.wisc.edu/condor/dagman [22/02/2011].

Deelman, E. and et al. (2006). Managing Large-Scale Workflow Execution from Resource Provisioning to Provenance tracking: The CyberShake Example. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*.

Hollingsworth, D. (1995). *The Workflow Reference Model*. Workflow Management Coalition.

Liu, D., Law, K. H., and Wiederhold, G. (2002). Dataflow Distribution in FICAS Service Composition Infrastructure. In *Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems*.

Martin, D., Wutke, D., and Leymann, F. (2008). A Novel Approach to Decentralized Workflow Enactment. *EDOC '08. 12th International IEEE Conference on Enterprise Distributed Object Computing*, pages 127–136.

Oinn, T. and et al (2004). Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA. ACM.

Sulakhe, D., Rodriguez, A., Wilde, M., Foster, I. T., and Maltsev, N. (2008). Interoperability of GADU in Using Heterogeneous Grid Resources for Bioinformatics Applications. *IEEE Transactions on Information Technology in Biomedicine*, 12(2):241–246.

Taylor, I., Shields, M., Wang, I., and Philp, R. (2003). Distributed P2P Computing within Triana: A Galaxy Visualization Test Case. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, pages 16–27. IEEE Computer Society.

Taylor, I. J., Deelman, E., Gannon, D. B., and Shields, M., editors (2006). *Workflows for e- Science: Scientific Workflows for Grids*. Springer-Verlag.

Wieland, M., Gorlach, K., Schumm, D., and Leymann, F. (2009). Towards reference passing in web service and workflow-based applications. In *Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*, pages 109 –118.