Eliminating The Middleman: Peer-to-Peer Dataflow

Adam Barker
National e-Science Centre
University of Edinburgh
a.d.barker@ed.ac.uk

Jon B. Weissman
University of Minnesota,
Minneapolis, MN, USA.
jon@cs.umn.edu

Jano van Hemert
National e-Science Centre
University of Edinburgh
j.vanhemert@ed.ac.uk

ABSTRACT

Efficiently executing large-scale, data-intensive workflows such as Montage must take into account the volume and pattern of communication. When orchestrating data-centric workflows, centralised servers common to standard workflow systems can become a bottleneck to performance. However, standards-based workflow systems that rely on centralisation, e.g., Web service based frameworks, have many other benefits such as a wide user base and sustained support.

This paper presents and evaluates a light-weight hybrid architecture which maintains the robustness and simplicity of centralised orchestration, but facilitates choreography by allowing services to exchange data directly with one another. Furthermore our architecture is standards compliment, flexible and is a non-disruptive solution; service definitions do not have to be altered prior to enactment. Our architecture could be realised within any existing workflow framework, in this paper, we focus on a Web service based framework.

Taking inspiration from Montage, a number of common workflow patterns (sequence, fan-in and fan-out), input to output data size relationships and network configurations are identified and evaluated. The performance analysis concludes that a substantial reduction in communication overhead results in a 2–4 fold performance benefit across all patterns. An end-to-end pattern through the Montage workflow results in an 8 fold performance benefit and demonstrates how the advantage of using our hybrid architecture increases as the complexity of a workflow grows.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Performance.

Keywords

Decentralised orchestration, workflow optimisation.

1. INTRODUCTION

Efficiently executing large-scale, data-intensive workflows common to scientific applications must take into account the volume and pattern of communication. For example, in Montage [7] an all-sky mosaic computation can require between 2–8 TB of data movement. Standard workflow tools based on a centralised enactment engine, such as Taverna [19] and OMII BPEL Designer [18] can easily become a performance bottleneck for such applications, extra copies of the data (*intermediate data*) are sent that consume network bandwidth and overwhelm the central engine. Instead, a solution is desired that permits data output from one stage to be forwarded directly to where it is needed at the next stage in the workflow. It is certainly possible to develop an optimised workflow system from scratch that implements this kind of optimisation. In contrast workflow systems based on concrete industrial standards offer a different set of benefits: they have a much larger and wider user base, which allows the leverage of a greater availability of supported tools and application components. This paper explores the extent to which the benefits of each approach can be realised. Can a standards-based workflow system achieve the performance optimisations of custom systems and what are the trade-offs?

1.1 Orchestration and Choreography

There are two common architectural approaches to implementing workflow; *service orchestration* and *service choreography.* Service orchestration describes how services can interact at the message level, with an explicit definition of the *control flow* and *data flow.* Orchestrations can span multiple applications and/or organisations, and 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)* [15] is the current defacto standard way of orchestrating Web services.

Service choreography on the other hand is more collaborative in nature. A choreography model describes a peer-to-peer collaboration between a collection of services in order to achieve a common goal. 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)* [9] is an XML-based language proposed for choreography. Currently this language is in the W3C candidate recommendation stage and there are no concrete implementations.

This paper presents a hybrid solution that "eliminates the middle man" by adopting an orchestration model of central control, but a choreography model of optimised distributed data transport. Our architecture could be realised within any existing workflow framework, even custom systems. In this paper, we focus on a Web service based implementation for the evaluation, a widely-promoted standard for building distributed workflow applications based on a suite of simple standards: XML, WSDL, SOAP, etc.

To explore the benefits of the hybrid approach for data-intensive applications, a set of workflow patterns and input-ouput relationships common to scientific applications (e.g. Montage) are used in isolation and combination. The performance analysis concludes that a substantial reduction in communication overhead results in a 2–4 fold performance benefit across all patterns. An end-to-end pattern through the Montage workflow demonstrates how the advantage of using the proxy architecture increases when patterns are used in combination with another, resulting in a 8 fold performance benefit. This paper does not address the performance limitations inherent in SOAP, an issue well addressed by other groups [5], [1].

2. SCIENTIFIC WORKFLOW PATTERNS

To identify data-centric workflow patterns, the Montage application has been used as an exemplar. It is representative of a class of large-scale, data-intensive scientific workflows. Montage constructs custom "science-grade" astronomical image mosaics from a set of input image samples [7]. The inputs to the workflow include the images in standard FITS format (a file format used throughout the astronomy community), and a "template header file" that specifies the mosaic to be constructed. The workflow can be thought of as having three parts, including re-projection of each input image to the coordinate space of the output mosaic, background rectification of the re-projected images, and co-addition to form the final output mosaic [4]. A typical montage workflow is depicted in Figure 1. This workflow consists of the following six components (with input-output relationships listed):

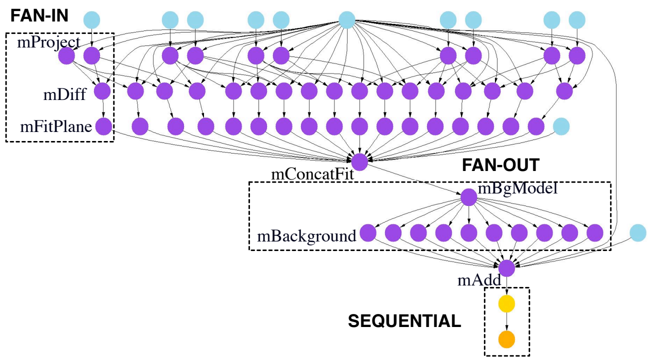


Figure 1: Montage Use-case scenario.

1. mProject: Re-projects a single image to the coordinate system defined in a header file (output = input)
2. mDiff/mFitPlane: Finds the difference between two images and fits a plane to the difference image (output = 15–20 % of a typical image for each image triplet)
3. mConcatFit: A simple concatenation of the plane fit parameters from multiple mDiff/mFitPlane jobs into a single file (see 4)
4. mBgModel: Models the sky background using the plane fit parameters from mDiff/mFitPlane and computes planar corrections for the input images that will rectify the background across the entire mosaic (output = a subset of inputs are output from mConcatFit and mBgModel)
5. mBackground: Rectifies the background in a single image (output = input)
6. mAdd: Co-adds a set of reprojected images to produce a mosaic as specified in a template header file (output = 70–90 % the size of inputs put together)

Montage illustrates several features of data-intensive scientific workflows. First, Montage can result in huge dataflow requirements. For example, a small input file is 1.5 MB and a small Montage application can consist of hundreds of input files, a larger problem, 10K–100K image files, all input in the mProject phase. The intermediate data can be 3 times the size of the input data. And a big problem, e.g. an all-sky mosaic can result in 2-8 TB of data. Such a problem might be run daily. Second, Montage contains workflow patterns common to many scientific applications:

1. Fan-in: e.g. mDiff/mFitPlane → mConcatFit
2. Fan-out: e.g. mBgModel → Background
3. Sequential: e.g. mConcat → mBgModel

Large-scale scientific workflows such as Montage may also have significant computational requirements that must be considered in deployment. In this paper, we consider optimisation of workflow patterns as representative of a class of large-scale data-intensive scientific workflows. We focus only on the orchestrations and techniques required to reduce the cost of communication, assuming the computational resources for executing the workflow have been identified.

3. HYBRID ARCHITECTURE

Currently most research has focused on designing languages for 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, examples of which can be found in the Business Process Modelling community through implementations of BPEL, in the Life Sciences through Taverna [19] and in the computational Grid community through Pegasus [4], Triana[14] and Kepler [13]. Choreography, although an established concept is a less well researched and implemented architecture, due to the complexity of message passing between distributed, concurrent processes.

To address the challenges posed by scientific workflow patterns both in size and structure, we propose a hybrid workflow architecture based on *centralised control flow, distributed data flow* [11]. 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, in a peer-to-peer fashion. 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.

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. Our proxy is a lightweight, non-intrusive piece of middleware, which provides a gateway and standard API to Web service invocation. This API contains the following operations: `invoke`, `stage`, `returnData`, `flushTempData`, `addService`, `removeService`, `listOps`, `listOpParams`, `listOpReturnType`, and `listServices`. Full details of the API and implementation of the proxy can be found in a complementary paper [2].

Proxies are installed as "near" as possible to enrolled Web services; by near we mean preferably on the same Web server or network domain, so that communication between a proxy and a Web service happens over a local 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, see the top and middle of Figure 2.

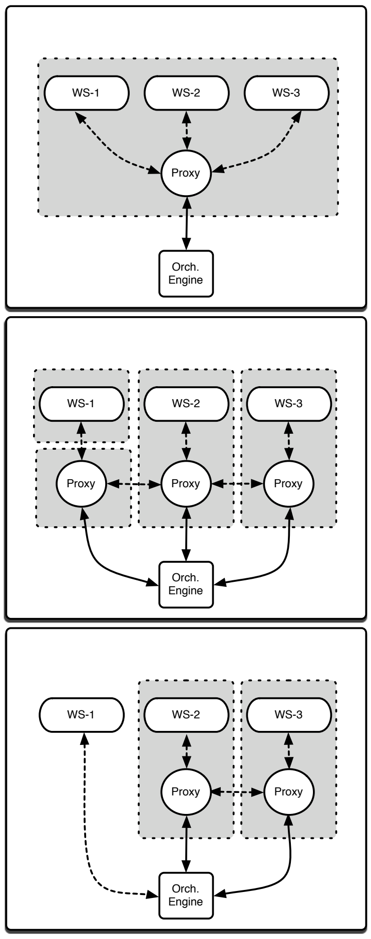


Figure 2: 1:N (top), 1:1 (middle), mixed (bottom).

To realise a centralised control flow, distributed data flow model, proxies are controlled by a centralised orchestration engine, 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 required for the next service invocation; this allows the orchestration engine to monitor the progress and make changes to the execution of a workflow.

Proxies themselves are exposed through a WSDL interface, allowing them to be built into workflows or higher level applications, like any other Web service. This means that workflows can be constructed from a combination of proxies and vanilla Web services, illustrated by the bottom of Figure 2. Unlike a pure choreography model, our architecture allows integration with centralised workflow systems making it easier to detect and handle failures. Furthermore the architecture offers the following software engineering advantages:

• **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. 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 near as possible to the Web services it is enabling.

3.1 Web Service Based Implementation

The Web service based proxy architecture is available as an Open Source toolkit, which is implemented using a combination of Java and the Apache Axis Web services toolkit [16]. A proxy is extremely simple to install and configure, it can be dropped into an Axis container running on a Tomcat server and can then be configured remotely. No specialised programming is needed 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, which deals with it in a First-In-First-Out (FIFO) order. Results from Web service invocations are tagged with a UID reference and stored at a proxy by writing the results to disk. Proxies are made available through a standard WSDL interface.

3.2 Example Application

The proxy architecture is most effectively illustrated through an example. Referring back to the Montage scenario, Figure 3 illustrates how our hybrid architecture can be applied to a section of the Montage scenario, in this case the sequence pattern discussed in Section 2. Active components in the scenario are coloured grey. In Figure 3 and the re-

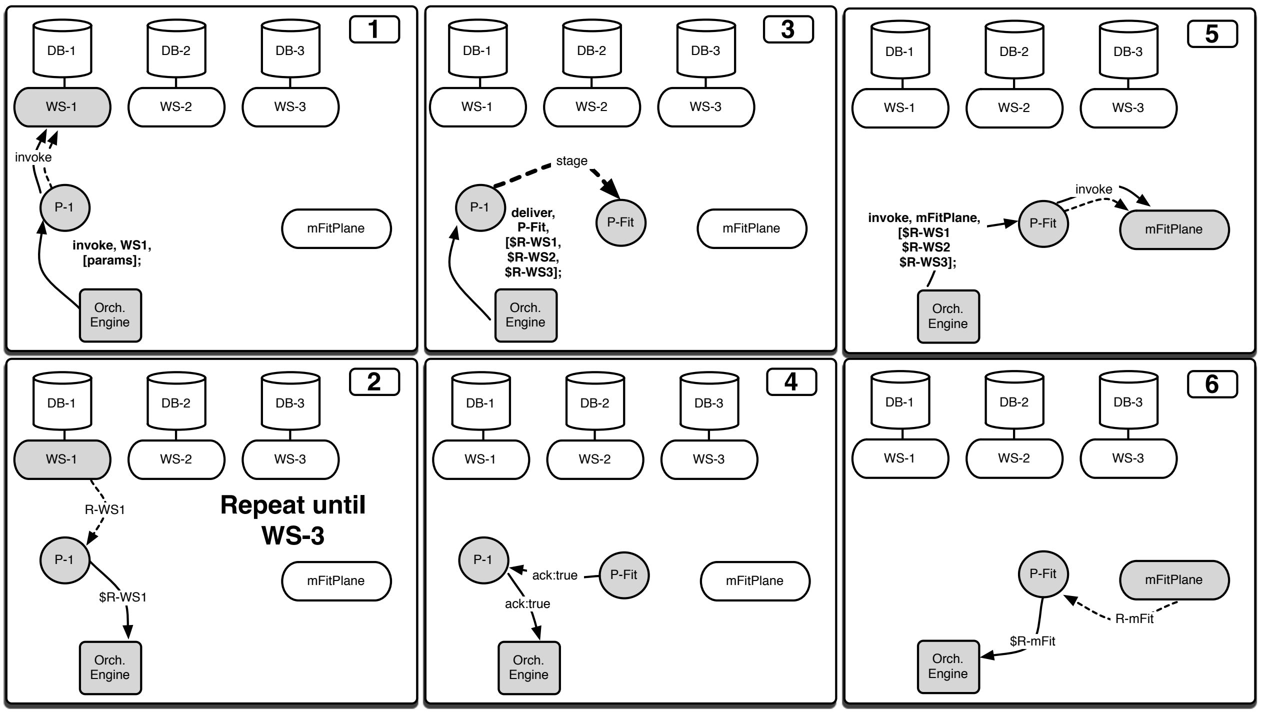


Figure 3: Example scenario - proxy architecture applied to the Montage sequential pattern using 4 services.

maining diagrams within this paper, control flow is displayed as a *solid black line* and data flow as a *dashed black line*. The workflow is described using a standards based workflow language (e.g. BPEL) and is enacted by a centralised orchestration engine. It is important to note that the choice of workflow language used to coordinate the proxies is entirely based on a user's preference and does not 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 pattern 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, passing the name of the Web service, port type and operation to be invoked, along with any required input parameters. 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 disk space to store the results. Instead of the proxy directly passing the data back to the orchestration engine, a reference to the data, `$R-WS1` is returned. In a standard orchestration scenario the results of the Web service invocation would have first been moved to the orchestration engine and then moved to where they are needed at the analysis Web service. However, as the proxy has been installed on the same server as `WS-1`, the data can be transferred locally between the proxy and the Web service and did not have to move over a Wide Area network, effectively saving a Wide Area hop. The only data returned to the orchestration engine was a reference to the output of the service invocation, `$R-WS1`. This process is repeated for `WS-2` and `WS-3` which could be served through the same proxy or an independent proxy, addressed by Figure 2.

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

• **Phases 5–6:** The final stage in the workflow pattern requires using the output from the first three services as input to the mFitPlane 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 data, which is required as input, in this case `$R-WS1, $R-WS2, $R-WS3`. The proxy then moves the data across the local network and invokes the operation which has been specified as input. The output, `R-mFit` is again stored locally on the proxy and a reference `$R-mFit` is passed back to the orchestration engine. The orchestration engine can then retrieve the actual data from the proxy when necessary.

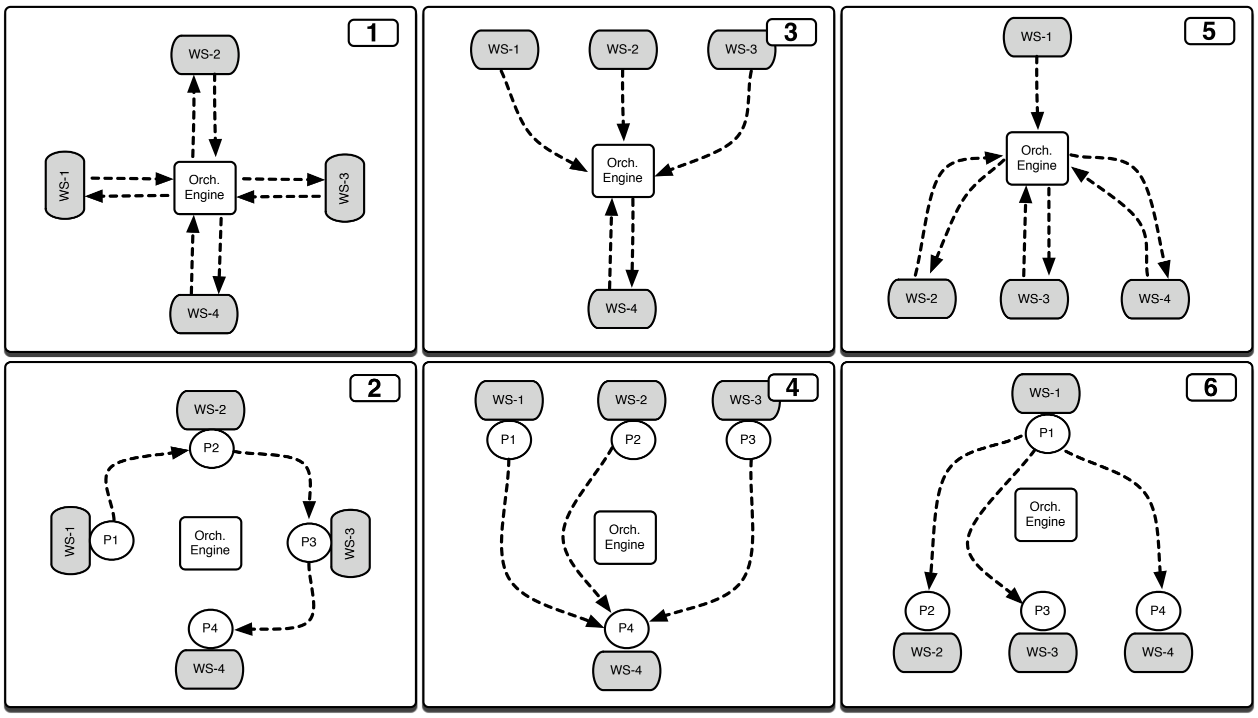


Figure 4: Dataflow in the sequential (first column), fan-in (second column) and fan-out (third column) patterns for the centralised architecture using vanilla services (1,3,5) and the proxy architecture (2,4,6). This example shows 4 services, all services are remote and all proxies are installed on the same server as the service they are invoking.

4. PERFORMANCE ANALYSIS

To verify our hypothesis we perform a set of performance analysis tests where our centralised control flow, distributed data flow proxy architecture is evaluated against a more traditional centralised control flow, centralised data flow orchestration engine.

4.1 Experiment Description

Taking inspiration from the Montage workflow, we perform tests with the patterns common to many scientific applications (sequential, fan-in and fan-out) both in isolation and in a combination. Furthermore, we show best-case and worst-case performance of our proposed architecture with respect to the location of the engine relative to the proxies. Throughout this paper we maintain the input-output data ratios discussed in Section 2. With reference to Figure 4, the patterns have been configured as follows:

• **Sequential pattern:** This pattern involves the chaining of services together, where the output of one service invocation is used directly as input to another. Once a service receives input data, its output is calculated by increasing the size of that input data by 20%, e.g. if the service receives 5Mb of data as input, 6Mb is returned as output. There is a snowball effect whereby the size of the data being transferred is increased after each service invocation. The configuration for this pattern on a fully centralised architecture is illustrated by phase 1 of Figure 4, and the configuration using our proxy architecture is illustrated by phase 2 of Figure 4.

• **Fan-in pattern:** The fan-in pattern explores what happens when data is gathered from multiple distributed sources, concatenated and sent to a service acting as a sink. Multiple services are invoked with a control flow (no data is sent) message asynchronously, in parallel, a block of data is then returned as output. Once data has been received from all enrolled services it is concatenated and sent to the sink service as input, where 20% of that input is returned as output. The configuration for this pattern using a fully centralised architecture is illustrated by phase 3 of Figure 4 and the configuration using our proxy architecture is illustrated by phase 4 of Figure 4.

• **Fan-out pattern:** This pattern is the reverse of the fan-in pattern, here the output from a single source is sent to multiple sinks. An initial service is invoked with a control flow message (again no actual data is sent), the service returns a block of data as output. This data is then sent, asynchronously in parallel to multiple services as input, each service returns as output the same size block of data it received as input. The configuration for this pattern using a fully centralised architecture is illustrated by phase 5 of Figure 4 and the configuration using our proxy architecture is illustrated by phase 6 of Figure 4.

For each of the workflow patterns: sequence, fan-in and fan-out the time taken for the pattern to complete is recorded (in milliseconds) as the size of the input data (in Megabytes) is increased; for the sequential pattern this means the size of the file *sent* to the first service, for the remaining patterns this means the size of the input file *returned* by the first service. The number of services involved in each of the patterns range from 3 to 17, this takes into account the lower bound (mProject → mDiff) and upper bound (mFitPlane → mConcatFit) limits of the Montage workflow scenario discussed in Section 2.

The configuration of our experiments mirror that of a typical workflow scenario, where collections of physically distributed services need to be composed into a higher level application. For each combination of input size, number of services and pattern type the experiment has been run independently 100 times over a cluster of distributed Linux machines. Wherever we report the time elapsed in milliseconds, 99% confidence intervals are included for each data point; some of these intervals are so small they are barely visible. Each line on the Figure 5, 6 and 7 displays the mean speedup ratio of each workflow pattern as the size of the input file increases. The mean speedup ratio is calculated by taking the average elapsed time (of 100 runs) for a vanilla (non-proxy, fully centralised) run of a workflow pattern and dividing it by the average elapsed time (of 100 runs) using our proxy architecture. The number of services involved is independent of the ratio as we have taken the mean ratio for all combinations of services (i.e. running the experiment iteratively on 3 to 17 services) from our scaling experiments[1].

In order to explore locality, the placement of the orchestration engine is also taken into consideration, displayed on each graph are four sub-experiments, in descending order according to the graphs the following has been plotted:

- **Remote best-case:** The orchestration engine is entirely remote to the services/proxies it is invoking, by remote we mean that the orchestration engine has to connect over a Wide Area network. It is the best-case as the final results are stored on the proxy and not returned to the orchestration engine. The best-case scenario is realistic as often individual patterns form only a small piece of a larger workflow as highlighted by the Montage scenario.
- **Remote worst-case:** In this sub-experiment the orchestration engine is again remote but the final output data of the workflow pattern are not stored at the proxy but sent back to the orchestration engine.
- **Local best-case:** The orchestration engine is deployed locally (i.e. on the same network) as the services/proxies it is invoking. The best-case represents the scenario where the final output from the pattern execution is stored within a proxy.
- **Local worst-case:** The final sub-experiment represents the case where the orchestration engine is again local but the final output of the pattern is sent back to the orchestration engine.

The input and output data in all the experiments are Java byte arrays passed around using SOAP. To prevent the data processing from influencing our evaluation, it has not been accounted for in the performance analysis tests.

4.2 Analysis of the Results

A collective summary of the performance analysis experiments is presented in Table 1. Displayed on each row is the pattern type, the corresponding experiment configuration, i.e. where the orchestration is and how the proxy behaves (best/worst-case), along with the mean speedup ratio, standard deviation, minimum and maximum speedup ratios. The end-to-end pattern is discussed in Section 4.3

The performance analysis tests verify our hypothesis that

[1]As an example the Appendix displays the elapsed time of the sequence, fan-in and fan-out workflow patterns using 4 distributed services when the orchestration engine is remote.

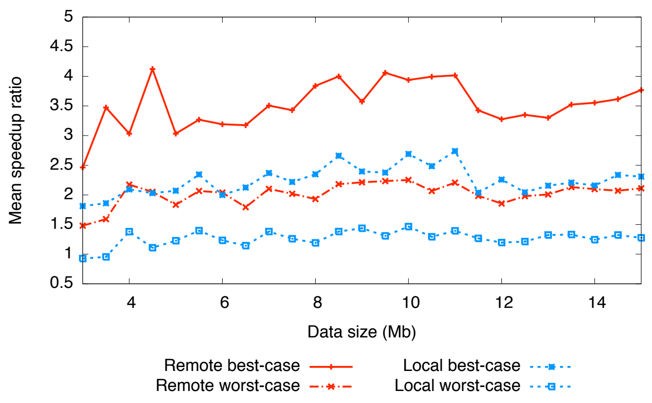


Figure 5: Sequential pattern - mean speedup ratio

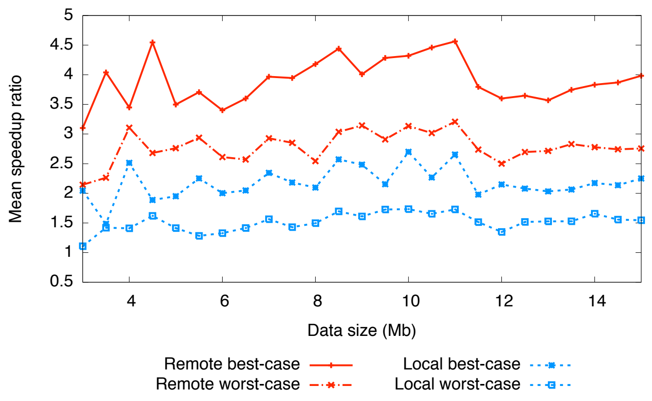


Figure 6: Fan-in pattern - mean speedup ratio

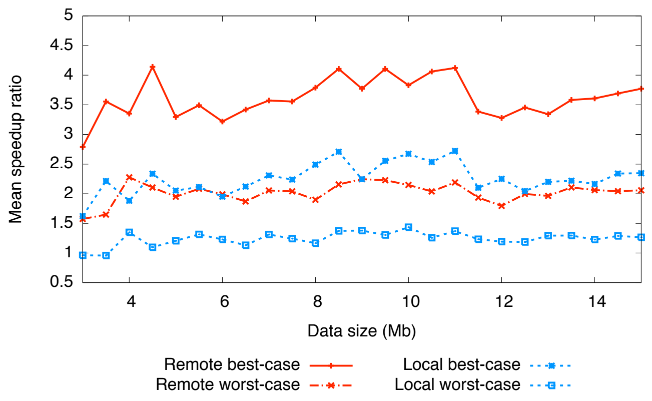


Figure 7: Fan-out pattern - mean speedup ratio

Table 1: Overview of performance analysis.

Pattern	Config	Mean	S Dev	Min	Max
Sequence	Local best	**2.21**	0.33	1.40	2.84
	Local worst	**1.29**	0.14	0.93	1.51
	Remote best	**3.47**	0.54	1.83	4.41
	Remote worst	**2.03**	0.18	1.48	2.28
Fan-in	Local best	**2.18**	0.32	1.25	2.74
	Local worst	**1.52**	0.19	0.97	1.81
	Remote best	**3.88**	0.53	2.23	4.97
	Remote worst	**2.83**	0.27	2.14	3.41
Fan-out	Local best	**2.25**	0.34	1.19	2.88
	Local worst	**1.26**	0.13	0.96	1.49
	Remote best	**3.61**	0.51	2.13	4.94
	Remote worst	**2.07**	0.21	1.57	2.63
End-to-End	Remote worst	**8.18**	0.94	5.58	9.86

when services are subscribed to our proxy architecture the execution time of common, isolated workflow patterns significantly decreases.

The locality experiments confirm that the most dramatic benefit occurs when the orchestration engine is connected to the services/proxies through a Wide Area network. To quantify, the worst-case remote configuration, patterns saw an average performance benefit of between 2.03 and 2.83 times and in the best-case remote configuration patterns an average performance benefit of between 3.47 and 3.88 times, with the fan-in pattern showing the largest speedup.

A surprising result of our experimentation is that even when the orchestration engine is deployed on the same network as the services/proxies it is invoking (i.e. all communication is local) there is a benefit to using the proxy architecture. In the worst-case local configuration patterns saw an average performance benefit of between 1.26 and 1.52 times and in the best-case local configuration patterns an average performance benefit of between 2.18 and 2.25 times.

To explain the results in relation to our proxy architecture, when using a fully centralised approach the intermediate data have to make a costly hop back to the orchestration engine before being again sent across the network to be used as input to the next service in the workflow. However, using the proxy architecture, intermediate data are stored at the proxy and sent directly to the next proxy which requires them as input, therefore for each input-output chain, one hop is avoided. In effect, this reduces the amount of intermediate data by 50%. This is, of course assuming that the proxy is installed as near as possible (i.e. on the same server or network) as the service it is invoking. This benefit is valid no matter where the orchestration is engine is placed, locally or remotely. Our locality experiments verify that even if workflows are orchestrated with locally deployed services the proxy architecture speeds up the overall execution time of a workflow pattern. However, as the orchestration engines moves further away, the hop any intermediate data has to make increases in cost and the benefit of using the proxy architecture increases accordingly. This explains why there is an increased benefit in the remotely deployed orchestration engine in relation to a locally deployed one. The difference in benefit is between 1.26 times and 1.70 times (mean remote-best − mean local-best across all patterns) in the best-case and between 0.74 times and 1.31 times (mean remote-worst − mean local-worst across all patterns) in the worst-case.

The results (Figure 5, 6 and 7) confirm our intuition that the co-plots are bounded by remote best-case (best performance) and local worst-case (worst performance) for all patterns. The other cases lie in-between and their relative position depends on the specific pattern. The results also show that the relative speed up is mildly sensitive to data size. This can be explained as the speed up ratio depends on the relative amount of data sent and the relative network bandwidth for the local and non-local cases, both of which are approximately constant. The later may have some SOAP/HTTP/TCP dependencies which likely accounts for the small variation seen. However, the raw differential performance between the proxy and vanilla version does scale with data size (see Appendix).

Although our experimentation is run at lower data sizes to Montage, patterns and input-output data relationships are maintained, this suggests that a similar performance benefit could be expected when scaling up the data injected into the workflow. Further experiments not discussed in this paper run over the PlanetLab [17] framework confirm that the ratios displayed in Table 1 match those obtained from running the same experiments over an Internet scale network.

4.3 End-to-End Execution

Section 4.1 and 4.2 discussed workflow patterns in isolation, however the sequential nature of the Montage workflow suggests that the optimisations of different workflow patterns will have an end-to-end cumulative performance benefit, e.g. speeding up the time to perform mConcatFit will allow mBgModel to execute earlier, and so on. In order to verify this hypothesis a path through the Montage workflow was investigated, this end-to-end pattern is illustrated in Figure 8.

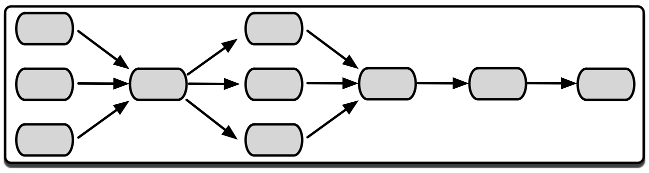


Figure 8: An end-to-end workflow, with a fan-in, fan-out followed by a series of sequential operations.

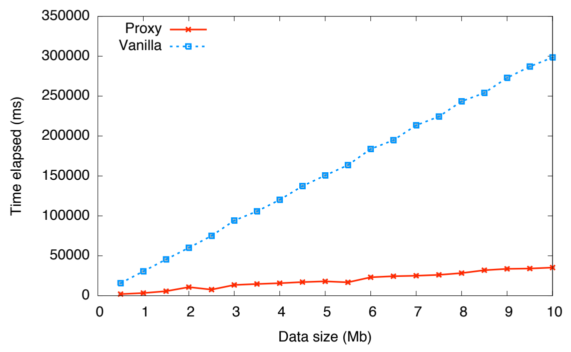


Figure 9: End-to-end pattern.

This combination of patterns comprises of the following steps, firstly a fan-in pattern that asynchronously in parallel gathers data from 3 different services, the output of which is sent to a further service which returns 20% of the input data as output data. This data is then sent asynchronously in parallel to 3 services which each return the same volume of output data as they received as input. The output data is concatenated and sent through a further 2 services in sequence, each return 50% of the data they received as input. These input-output data relationships mirror those found in the Montage scenario.

The end-to-end pattern displayed in Figure 8 is executed 100 times on our proxy architecture (using the worst-case, i.e. the final output data returns to the orchestration engine) and 100 times on a fully centralised orchestration engine with vanilla Web services, on both occasions the orchestration engine is remote. Figure 9 shows the results with a confidence interval of 99% where the x-axis displays the input data size in Megabytes and the y-axis displays the time taken in milliseconds to complete the workflow. The end-to-end execution results in a mean speedup of 8.18 times using the proxy architecture, confirming our hypothesis that the performance benefit increases when isolated patterns are placed together to form a larger workflow. This sample end-to-end execution demonstrates the concept, however this combination pattern itself would only form a small part of larger scientific workflows, such as Montage.

4.4 Break Even Point

Invoking a proxy has an overhead in that a call is first made to a proxy, which invokes the service on the orchestration engines behalf, writes the result to disk and then returns a reference to that data. As the previous performance analysis tests demonstrate what occurs on relatively large data sizes, it is important to highlight what happens when dealing with Kilobytes instead of Megabytes of data in order to determine the break even point, i.e. when using a proxy is preferable to a vanilla service invocation.

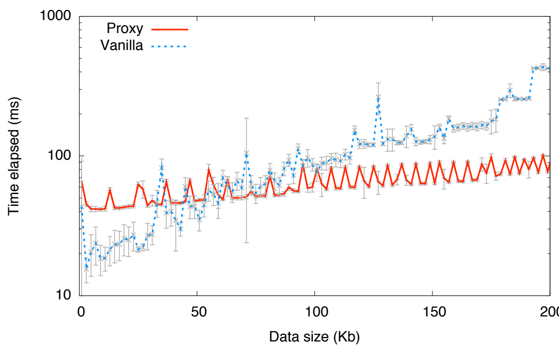


Figure 10: The overhead of invoking a proxy.

Figure 10 displays (with a 99% confidence interval) the average time it takes to make a single invocation to a vanilla Web service and obtain the result vs. an invocation to a proxy that invokes the service on the orchestration engines behalf and returns a reference to its data. From the results we conclude that due to the overhead of the proxy, when dealing with input data sizes of less than ∼120K of data the proxy architecture offers no performance benefit to vanilla Web services. Anything over ∼120K of data the proxy begins to speedup the execution time of the invocation. The proxy architecture is suited to larger scale workflows (such as Montage) and not workflows where very small quantities of intermediate data are passed around between services, i.e. typical scenarios in business.

5. RELATED WORK

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

5.1 FICAS Architecture

The Flow-based Infrastructure for Composing Autonomous Services or FICAS [12] is a distributed data-flow architecture for composing software services. 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. 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.

5.2 Service Invocation Triggers

Service Invocation Triggers [3] are also a response to the problem of centralised orchestration engines when dealing with large-scale data sets. Triggers collect the required input data before they invoke a service, forwarding the results directly to where the data is required. For this decentralised execution to take place, a workflow must be deconstructed into sequential fragments which contain neither loops nor conditionals and the data dependancies must be encoded within the triggers themselves.

The approach outlined by our paper and Service Invocation Triggers both rely on proxies to solve the problem of decentralised orchestration. While Triggers address the issue of decentralised control, to realise these benefits their architecture is based around a pure choreography model, which as discussed in 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.

5.3 Techniques in Data Flow Optimisation

• OGSA-DAI [8] middleware supports the exposure of data resources on to Grids and facilitates data streaming between local OGSA-DAI instances.

• Grid Services Flow Language (GSFL) [10] addresses some of the issues discussed in this paper in the context of Grid services, in particular services adopt a peer-to-peer dataflow model. However, individual services have to be altered prior to enactment, which is an invasive and custom solution, something avoided in our hybrid architecture.

• Graph-forwarding is a technique [6] applied to distributed Objects, allowing the results of an RPC to be forwarded to the next object to invoke instead of the invoking object.

6. CONCLUSIONS

This paper presented a light-weight hybrid architecture for executing large-scale data-centric workflows. Our architecture maintains the robustness and simplicity of centralised orchestration, but facilitates choreography by allowing services to exchange data directly with one another. Using Montage as a guide, a number of common workflow patterns and input-output relationships are evaluated in a Web services based framework. Although this paper discussed the hybrid architecture in a Web services context, it is a general architecture and can therefore be implemented using different technologies and integrated into existing systems. Furthermore our architecture is non-invasive to the Web services themselves.

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. The results indicate that substantial reduction in communication overhead results in a performance benefit of between 2.03 and 3.88 times. The advantage of using the proxy architecture increases if isolated patterns are used in combination with another, the end-to-end pattern demonstrates an 8 fold performance benefit.

Future directions include evaluating the benefits of our approach within other workflow frameworks and in other network environments (e.g. Wide Area, mobile) to assess the impact in different contexts. The analysis of additional applications to identify and evaluate other end-to-end workflow patterns is also planned. This architecture also opens up a rich set of additional optimisations with respect to proxy deployment which will be evaluated in future work.

7. ACKNOWLEDGEMENTS

Professor Daniel Katz from Louisiana State University provided detailed information about the structure and operation of Montage that was central to this paper.

8. REFERENCES

[1] Nayef Abu-Ghazaleh, Michael J. Lewis, and Madhusudhan Govindaraju. Differential Serialization for Optimized SOAP Performance. In *Proceedings of HPDC*, June 2004.

[2] Adam Barker, Jon. B. Weissman, and Jano van Hemert. Orchestrating Data-Centric Workflows. In *Proceedings of the Eighth IEEE International Symposium on Cluster Computing and the Grid*, 2008.

[3] 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.

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

[5] K. Devaram and D. Andresen. Differential Serialization for Optimized SOAP Performance. In *Proceedings of PDCS*, November 2003.

[6] 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.

[7] J. C. Jacob, D.S. Katz, and et. al. The Montage Architecture for Grid-Enabled Science Processing of Large, Distributed Datasets. In *Proceedings of the Earth Science Technology Conference*, June 2004.

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

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

[10] S. Krishnan, P. Wagstrom, and G. von Laszewski. GSFL: A Workflow Framework for Grid Services. Technical report, Argonne National Argonne National Laboratory, 2002.

[11] 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.

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

[13] B Ludascher and et al. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2005.

[14] Ian J. Taylor and et al. 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, 2003.

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

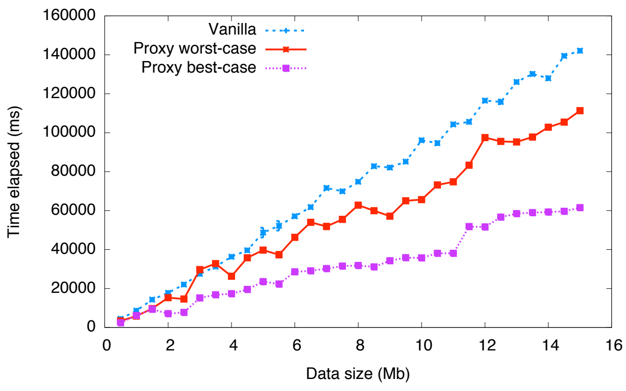
[16] Apache Axis: `http://ws.apache.org/axis`.

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

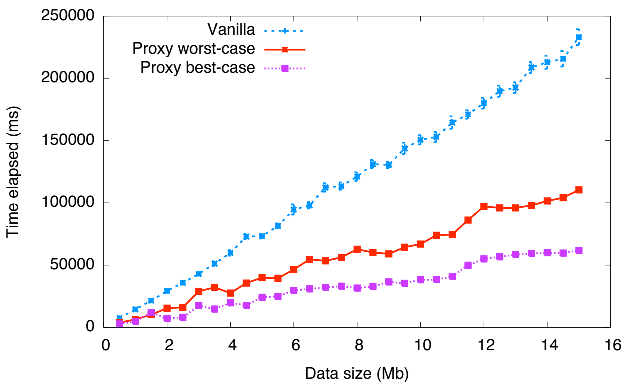
[18] Bruno Wassermann and et al. Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modelling. *Workflows for eScience - Scientific Workflows for Grids*, December 2006.

[19] 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.

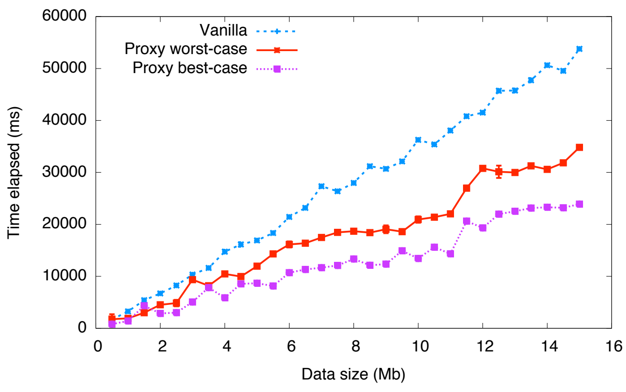
APPENDIX



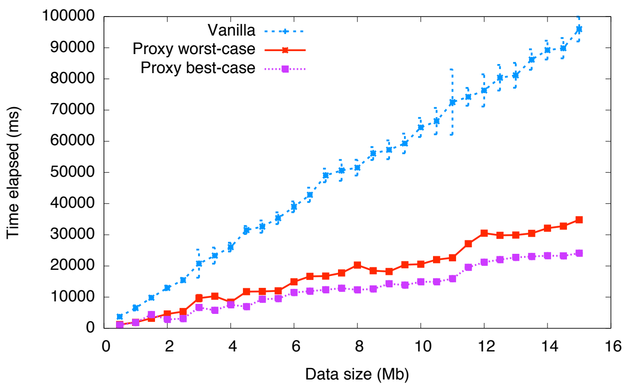
(a) Sequential pattern LOCAL orchestration engine



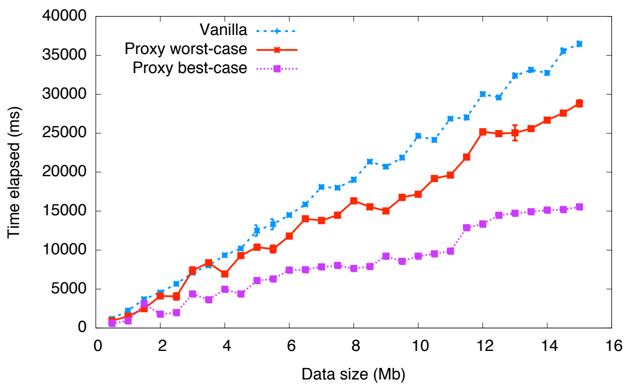
(b) Sequential pattern REMOTE orchestration engine



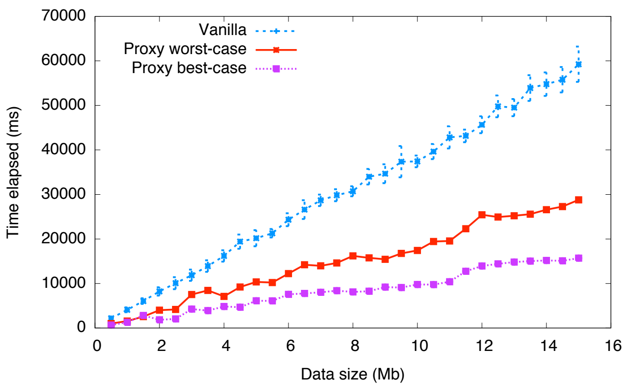
(c) Fan-in LOCAL orchestration engine



(d) Fan-in REMOTE orchestration engine



(e) Fan-out LOCAL orchestration engine



(f) Fan-out REMOTE orchestration engine

Figure 11: An example experiment, using 4 services, recording the average time it takes for each pattern to complete as the size of the input data increases. The x-axis display the size of the initial input file in Megabytes (Mb) and the y-axis displays the elapsed time of the workflow pattern in milliseconds (ms). In 11(a), 11(c) and 11(e) the orchestration engine is locally deployed, in 11(b), 11(d) and 11(f) the orchestration is remotely deployed.