The Benefits of Service Choreography for Data-Intensive Computing

Adam Barker
Dept. of Engineering Science,
University of Oxford, UK.
adam.barker@gmail.com

Paolo Besana,
David Robertson
School of Informatics,
University of Edinburgh, UK.
p.besana@sms.ed.ac.uk,
dr@inf.ed.ac.uk

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

ABSTRACT

As the number of services and the size of data involved in workflows increases, centralised orchestration techniques are reaching the limits of scalability. In the classic orchestration model, all data pass through a centralised engine, which results in unnecessary data transfer, wasted bandwidth and the engine to become a bottleneck to the execution of a workflow. Choreography techniques, although more complex to model offer a decentralised alternative and are the optimal architecture for data-centric workflows; data are passed directly to where they are required, at the next service in the workflow.

While orchestration is the dominant architectural approach, there are relatively few choreography languages and even fewer concrete implementations. This papers contributions are twofold. Firstly we argue the case for choreography in data-intensive computing, and demonstrate through workflow patterns the advantages in terms of scalability when a choreography architecture is adopted. Secondly we introduce the Light Weight Coordination Calculus (LCC), a type of process calculus used to formally define choreographies, and the OpenKnowledge framework, a choreography-based architecture, providing the functionality for peers to coordinate in an open peer-to-peer system. Through LCC and the OpenKnowledge framework we practically demonstrate how choreography can be achieved in a lightweight manner with a comparatively simple process language.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.11 [**Software Engineering**]: Software Architectures; H.4.1 [**Information Systems Applications**]: Workflow Management

General Terms

Algorithms, Design.

CLADE'09, June 9–10, 2009, Munich, Germany.

1. INTRODUCTION

Service-oriented architecture (SOA) is an architectural approach for the implementation and delivery of loosely coupled distributed services [26]. Although the concept of a service-oriented architecture is not a new one, this approach has seen wide spread adoption through the Web services approach, which has a set of basic, core standards (XML, WSDL, SOAP etc.) to facilitate service interoperability.

The core standards do not provide the rich behavioural detail which describes the role an individual service plays as part of a larger, more complex collaboration. This collaboration is often achieved through the use of workflow technologies. As defined by the Workflow Management Coalition [13], 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. Workflow can be described from the view of a single participant using *orchestration* or from a global perspective using *choreography.*

Web service orchestration enables Web services to be composed together in predefined patterns, described using an *orchestration language* and executed on an *orchestration engine.* Services themselves have no knowledge of their involvement in a higher level application and therefore need no alteration before enactment. Importantly, Web service orchestrations are described from the view of a *single participant* (which can be another Web service) and therefore a central process always acts as a controller to the involved services. Orchestration languages explicitly describe the interactions between Web services by identifying messages, branching logic and invocation sequences. The *Business Process Execution Language (BPEL)* [29] is an executable business process modelling language and is recognised as the current de-facto standard way of orchestrating Web services. BPEL has broad industrial support from companies such as IBM, Microsoft and Oracle, with concrete implementations.

Service choreography on the other hand is more collaborative in nature. A service choreography is a description of the *externally observable* peer-to-peer interactions that exist between services, therefore choreography does not rely on a central coordinator. A choreography model describes multi-party collaboration and focuses on message exchange; each Web service involved in a choreography knows exactly when to execute its operations and with whom to interact. A choreography definition can be used at design-time to ensure interoperability between a set of peer services from a

global perspective, meaning that all participating services are treated equally, in a peer-to-peer fashion. The *Web Services Choreography Description Language (WS-CDL)* [17] is an XML-based language that can be used to describe the common and collaborative observable behaviour of multiple services that need to interact in order to achieve a shared goal. WS-CDL is a W3C Candidate Recommendation. For a summary of the current orchestration and choreography languages refer to [1].

1.1 Paper Contributions and Overview

There is a shortfall in choreography research; the current primary focus by industry and to some degree academia is on orchestration techniques. To roughly quantify, according to Google Scholar: From 1990 until 2008 there were 13,000 orchestration related publications and 4,700 choreography related publications. There is a varied selection of orchestration languages and enactment frameworks, examples can be seen in the Business Process Modelling community through BPEL and life sciences community through Taverna [22].

This paper highlights the case for choreography in data-intensive computing, focusing in particular on the bottlenecks caused by centralised orchestration engines. Arguments for choreography are presented in Section 2. In Section 2.1 a number of simple workflow patterns focusing on data flow are introduced. These workflow patterns are used as the basis for discussion throughout the remainder of this paper. For each of the patterns (fan-in, fan-out and sequence) orchestrated and choreographed solutions are presented. We compare the approaches and demonstrate the savings in terms of data flow when individual patterns are choreographed.

Section 3.1 introduces the OpenKnowledge framework, a choreography-based architecture centred around peers in an open peer-to-peer system. The framework enables the publishing and enactment of interaction models, a formal specification of a choreography. Section 3.2 presents an overview of the Light Weight Coordination Calculus (LCC), a type of process calculus used to formally define interaction models. In order to demonstrate the OpenKnowledge framework and LCC, the fan-in pattern is implemented in Section 4. There exists a limited set of pure choreography languages, however Section 5 discusses each in turn, along with approaches for optimising Web service orchestration. Finally, conclusions are presented in Section 6.

2. THE CASE FOR CHOREOGRAPHY

Choreography, although an established concept, is a less well researched and implemented architecture than orchestration. In practice, the design processes and execution infrastructure for service choreography models are inherently more complex than orchestration; decentralised control brings a new set of challenges which are the result of message passing between distributed asynchronous, concurrent processes. However, although more complex, there are a number of arguments for adopting choreography.

From a software design perspective, orchestration is suitable when the goal is to build individual services or to service-enable existing applications. However during the early phases of service design the emphasis lies not on building individual services but rather on how groups of services work together, by identifying collections of potential services and understanding and analysing their interactions; at this early stage in the design process engineers require a global view of how Web services interact with one another; choreography provides just these tools and is a description of multi-party collaboration.

Choreography is an unambiguous way of describing the relationships between services in a global peer-to-peer collaboration, without requiring orchestration at all. Each party takes an equal, predefined and pre-agreed role in the choreography, this removes the scenario where businesses are interacting over a shared task but one organisation has control over another by orchestrating their services.

Centralised control through an orchestration engine is a valid solution for scenarios found in e-Commerce, where relatively small quantities of *intermediate data* (when output from one service invocation is directly, with no alteration, used as input to another) are moved between services in a workflow. However, centralised servers make less sense when dealing with data centric workflows (GBs/TBs), common to scientific applications. Passing large quantities of intermediate data through a centralised orchestration engine results in unnecessary data transfer and wasted bandwidth, overloading the engine and thereby decreasing the performance of a workflow.

2.1 Expanding the Scalability Argument

As with software design patterns, workflow patterns refer to recurrent problems and proven solutions in the development of workflow applications. There is a large body of workflow patterns research detailing a comprehensive set of patterns from both a control flow and data flow perspective, the most prevalent being the work by van der Aalst and Hofstede et al. [31]. and the Service Interaction Patterns set by Barros et al. [5], a collection of thirteen recurring patterns derived from insights into business-to-business transaction processing.

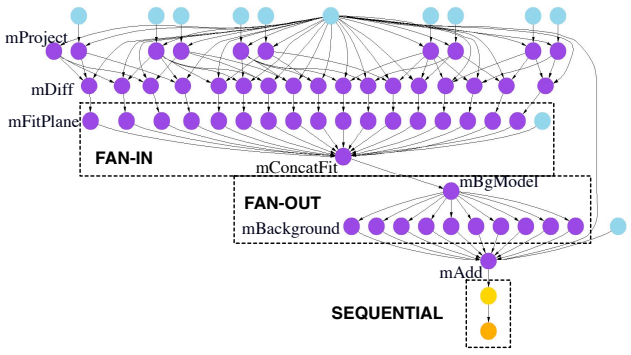


Figure 1: Pattern extraction: Montage use-case scenario.

Of particular interest to this paper are patterns which affect data flow and not control flow. Figure 1 illustrates the Montage workflow, a benchmark in the High Performance Computing community and representative of a class of large-scale, data-intensive workflows. Montage constructs custom "science-grade" astronomical image mosaics from a set of input image samples [14]. Montage illustrates several features of data-intensive scientific workflows. Montage can result in huge data flow requirements. The intermediate

data can be 3 times the size of the input data, e.g. an all-sky mosaic can result in 2-8 TB of data movement. Such a problem might be run daily. Montage is essentially a Directed Acyclic Graph (DAG).

With reference to Figure 1 we can identify a number of simple patterns:

- *Fan-in*: Involves mapping multiple sources to a single sink (N:1 relationship), e.g. mFitPlane → mConcatFit.
- *Fan-out*: The reverse pattern of fan-in, data from a single source is sent to multiple sinks (1:N relationship), e.g. mBgModel → Background.
- *Sequence*: This pattern involves the chaining of services together, where the output of one service invocation is used directly as input to another, i.e. serially (1:1 relationship). The data flows as a pipeline with no data transformations, e.g. mConcat → mBgModel.

In order to highlight the scalability and optimised data flow argument for each pattern we highlight in terms of data flow the orchestration scenario, left hand column of Figure 2 and the choreography scenario, right hand column of Figure 2. Diagrams are drawn in UML Collaboration format. Data sizes are shown in Megabytes (Mb) and are used for illustrative purposes only. The following two equations are used to calculate the data flow on a per pattern relationship, i.e. fan-in (N:1), fan-out (1:N) and sequence (1:1).

Equation 1 is used to calculate the total data flow for a given pattern using orchestration. Equation 2 is used to calculate the total data flow using choreography. $\mathtt{n}$ represents the number of sources and $\mathtt{i}$ represents the index of a source service $\mathtt{for}\ 1 \leq \mathtt{i} \leq \mathtt{n}$. $\mathtt{in_i}$ represents the input size in Mb to a source service $\mathtt{i}$. $\mathtt{out_i}$ represents the output size in Mb to a source service $\mathtt{i}$. $\mathtt{m}$ represents the number of sinks and $\mathtt{k}$ represents the index of a sink service $\mathtt{for}\ 1 \leq \mathtt{k} \leq \mathtt{m}$. $\mathtt{out_k}$ represents the output size in Mb to a sink service $\mathtt{k}$. We do not propose that these equations are general purpose, they have been included to highlight the difference in data flow for our particular configuration of patterns.

$$\sum_{i=1}^{n}\left((in_i + (m+1) \times out_i) + \sum_{k=1}^{m}(out_k)\right) \quad (1)$$

$$\sum_{i=1}^{n}\left((in_i + m \times out_i) + \sum_{k=1}^{m}(out_k)\right) \quad (2)$$

2.1.1 Fan-in pattern

Three services $\mathtt{source1} - \mathtt{source3}$ are queried for data, these data are concatenated and sent to a final service $\mathtt{processor}$ for analysis, which returns 10% of the input data size as output. In our example, $\mathtt{n} = 3$ and $\mathtt{m} = 1$. Using Equation 1, to orchestrate the fan-in pattern (Figure 3(a)) involves a total data flow of 630Mb. Using equation 2, to choreograph (Figure 3(b)) a total of 330Mb.

2.1.2 Fan-out pattern

A service $\mathtt{source}$ is queried for data, which are sent asynchronously in parallel to three services $\mathtt{processor1} - \mathtt{processor3}$, for analysis. Each service then returns 10% of the input it was provided. In our example, $\mathtt{n} = 1$ and $\mathtt{m} = 3$. Using Equation 1, to orchestrate (Figure 3(c)) the fan-out pattern involves 430Mb of data flow. Using Equation 2, to choreograph (Figure 3(d)) the pattern involves 330Mb of data flow.

2.1.3 Sequence pattern

A query is made to a data $\mathtt{source}$, whose output is chained serially through three analysis services $\mathtt{processor1} - \mathtt{processor3}$, each service returns half of the input it receives.

As the sequence pattern defines a 1:1 relationship, we must use Equation 1 for every output → input chain to calculate the data flow for the entire workflow: i.e. $\mathtt{source}$ → $\mathtt{processor1}$ and $\mathtt{processor2}$ → $\mathtt{processor3}$ for Figure 3(e). Therefore $\mathtt{n} = 1$ and $\mathtt{m} = 1$ for two calculations and the total data flow to orchestrate the pattern is 362.5Mb. Using Equation 2 to choreograph (Figure 3(f)) involves 187.5Mb of data flow. Note, for the sequence choreography calculation, only the initial data flow to the first service in the chain is used to calculate the total data flow. Therefore the input and output to $\mathtt{source}$ and the output from $\mathtt{processor1}$ make up the first chain and the output from $\mathtt{processor2}$ and $\mathtt{processor3}$ make up the second chain.

2.2 Discussion

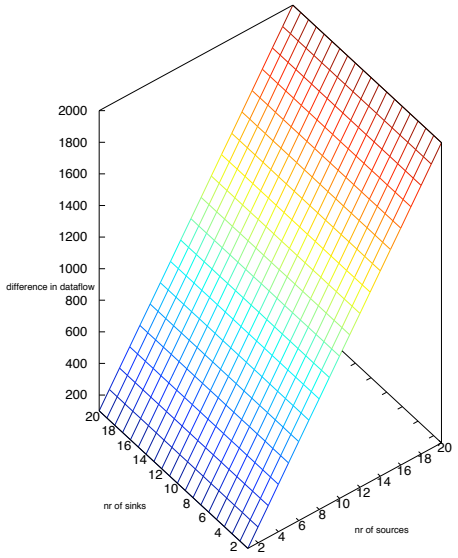
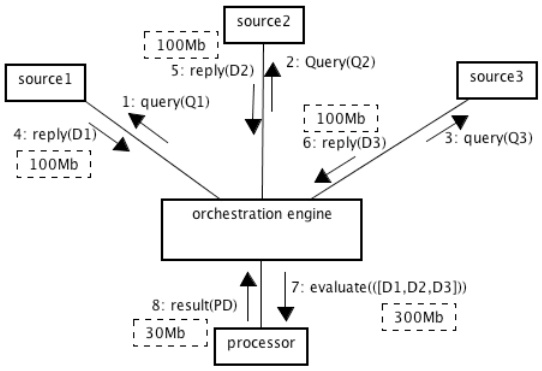
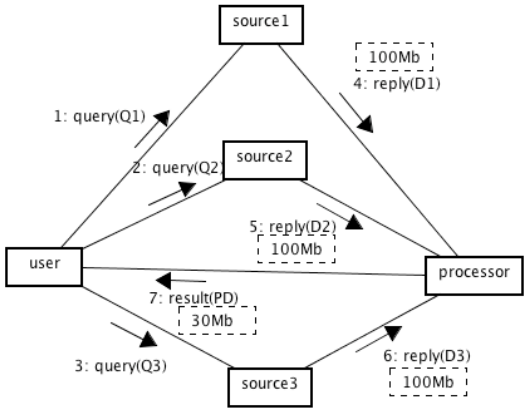


Figure 2: Difference in the data flow between orchestration and choreography increasing the number of sinks and sources.

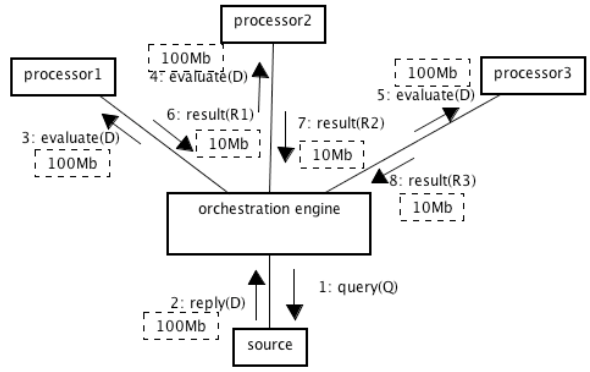
For each of the patterns we have demonstrated how the choreographed solution is optimal in terms of data flow. Choreography removes the notion of intermediate data. As an example, consider the fan-in pattern. Using standard orchestration the query results $(\mathtt{D1}, \mathtt{D2}, \mathtt{D3})$ from $\mathtt{source1} - \mathtt{source3}$ pass through the centralised orchestration engine and are then used as input to the $\mathtt{processor}$



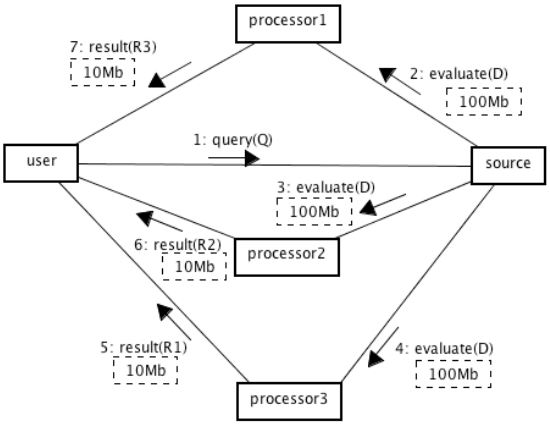
(a) Orchestrated Fan-in pattern



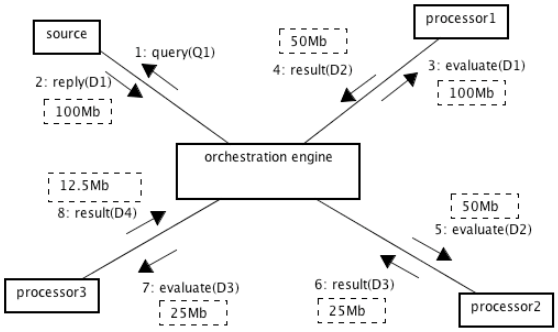
(b) Choreographed Fan-in pattern



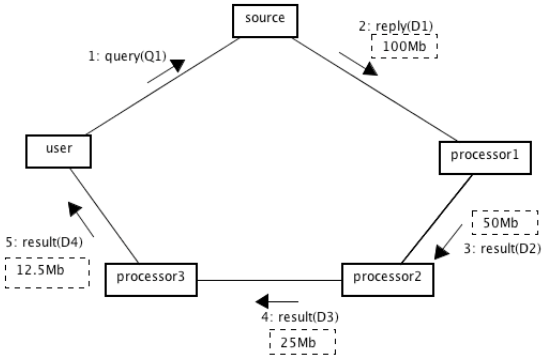
(c) Orchestrated Fan-out pattern



(d) Choreographed Fan-out pattern



(e) Orchestrated Sequence pattern



(f) Choreographed Sequence pattern

Figure 3: Workflow patterns: Fan-in, Fan-out, Sequence. Orchestration (left column) and Choreography (right column). We assume the `orchestration engine` is running on a user's desktop and is therefore equivalent to the `user` party in the choreography scenario. No data transformations take place and any control flow is ignored. Data size shown in Megabytes (Mb).

service, which analyses these data and returns the results (PD) back to the engine. Using choreography, the results of the queries (D1, D2, D3) from source1 − source3 are sent directly to where they are required, as input to the processor service. Once analysis is complete the results (PD) are sent back to the user, whom is remote from the distributed services. As the results from the queries (D1, D2, D3) do not have to pass through a centralised orchestration engine, the choreography approach removes three additional hops involving intermediate data. This optimises data flow by a total of 300Mb.

For every output → input chain, that is every output that is used directly as input to another service, orchestration adds an extra hop to the data flow as it passes through the orchestration engine. Totaling n extra hops for each pattern. This is reflected in Equation 1 through $(m + 1) \times out_i$ against $m \times out_i$ in Equation 2. The benefits of adopting a choreography model increase as the number of services and data sizes used in the workflow pattern increase. However, as Figure 2 shows, the greatest benefit is obtained as the number of sources n increases: for twenty sources and one sink, with the same input and output sizes used in the example, the difference between orchestration and choreography data flow is nearly 50% (2200Mb against 4200Mb); increasing the number of sinks does not change the difference, as it is proportional to the number of sources only.

3. ADDRESSING THE BOTTLENECK

In the context of our scalability discussion, this Section introduces a choreography-oriented language, the Light Weight Coordination Calculus (LCC) and its corresponding enactment framework, OpenKnowledge.

3.1 The Open Knowledge Framework

The OpenKnowledge kernel [27] provides the layer that assorted services and applications can use to interact using a choreography-based architecture. The core concept is the use of *shared interaction models* by different applications and service providers. These actors are the *participants* of the interactions, and they assume *roles* within them. In an interaction all roles have equal weight; in that respect they are *peers* in a peer-to-peer network and their behaviours and in particular their exchanges of messages are specified.

Interaction models are published by the authors on the *Distributed Discovery Service* (DDS), with a keyword-based description [18]. A peer that wants to perform some task, such as querying a database, providing access to data or data processing capabilities, searches for published interaction models (IMs) by sending a keyword-based query to the DDS. The DDS collects the published interaction models matching the description (the keywords are extended adding synonyms to improve recall) and sends back the list.

The peer then evaluates the requirements of the interaction model with its own capabilities. If they match, the peer will send back to the DDS the subscription to one of the interaction model's roles, advertising its intention to participate in one or more executions of the interaction. Peer's capabilities are provided by plug-in components, called *OKCs* (Open Knowledge Component). An OKC is a jar file, and its classes expose methods that implement some functionality. How the functionality is implemented is transparent: it can be an invocation to a web service, a call to a legacy application or it can be self contained and use only what is contained in the jar. OKCs can be published on the DDS and downloaded by any peer.

Figure 4 illustrates a snapshot of a deployed OpenKnowledge framework. Interaction Models IM_1, IM_2 and IM_3 are published on the DDS. In IM_1, role r1 is subscribed by peer P_1, role r2 is subscribed by P_2 and role r2 is subscribed by P_3 and P_4.

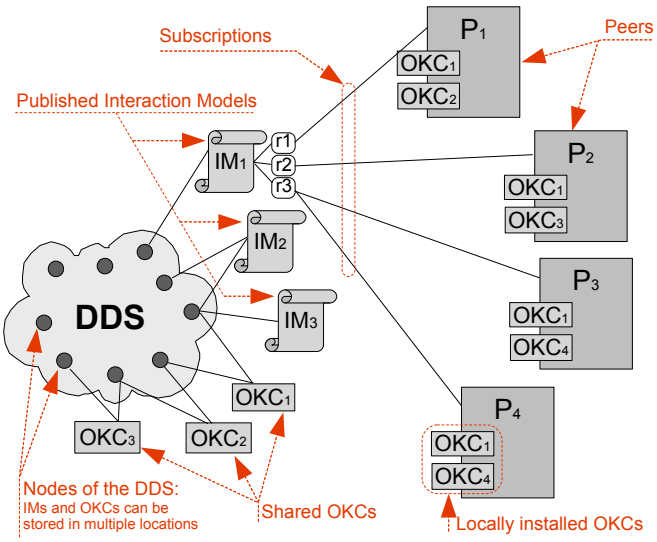


Figure 4: OpenKnowledge architecture

Once all roles are filled, the DDS randomly selects a peer in the network to act as coordinator for the interaction, and passes the interaction model together with the list of involved peers in order to bootstrap it. The coordinator first asks each peer to select the peers they want to interact with, forming a mutually compatible group of peers out of the replies. Once complete, enactment of the interaction model can begin.

The kernel has been developed in Java and it is less than 15Mb. It can be downloaded from the OpenKnowledge website: [30], where it is also possible to find more exhaustive information.

3.2 Lightweight Coordination Calculus

Interaction models are written in the *Lightweight Coordination Calculus (LCC)* [25], an executable choreography language based on process calculus.

$$
\begin{array}{rcl}
Model & := & \{Clause, \ldots\} \\
Clause & := & Role :: Def \\
Role & := & a\,(Type, Id) \\
Def & := & Role \mid Message \mid Def\ then\ Def \mid Def\ or\ Def \\
Message & := & M \Rightarrow Role \mid M \Rightarrow Role \leftarrow C \\
 & & \mid M \Leftarrow Role \mid C \leftarrow M \Leftarrow Role \\
C & := & Constant \mid P(Term, \ldots) \mid \neg C \mid C \wedge C \mid C \vee C \\
Type & := & Term \\
Id & := & Constant \mid Variable \\
M & := & Term \\
Term & := & Constant \mid Variable \mid P(Term, \ldots) \\
Constant & := & \text{lower case character, sequence or number} \\
Variable & := & \text{upper case character, sequence or number}
\end{array}
$$

Figure 5: LCC formal syntax

With reference to Figure 5, an interaction model in LCC is a set of clauses, each of which define how a role in an interaction must be performed. Roles are described by their type and by an identifier for the individual peer undertaking that role. Participants in an interaction take their initial *entry-role* and follow the unfolding of the clause specified using a combination of the sequence operator ('*then*') or choice operator ('*or*') to connect messages and changes of role. Messages are either outgoing to ('⇒') or incoming from ('⇐') another participant in a given role. Messages can have constraints: success or failure in satisfying them drives the unfolding of the interactions. In OpenKnowledge the constraints are matched to methods in the OKCs, the plug-in components introduced in the previous Section. During an interaction, a participant can take, sequentially, different roles and can recursively take the same role, for example when recursively processing a list. Message input/output or change of role is controlled by constraints defined using conjunction and disjunction.

$$
\begin{array}{l}
\mathtt{a}(\mathtt{r_1}, \mathbf{A}) :: \\
\left(\begin{array}{l}
\mathtt{msg_1}(\mathtt{Data}) \Rightarrow \mathtt{a}(\mathtt{r_2}, \mathtt{B}) \leftarrow \mathtt{obtain}(\mathtt{Data}) \\
\mathtt{then} \\
\mathtt{msg_2} \Leftarrow \mathtt{a}(\mathtt{r_2}, \mathtt{B})
\end{array} \right) \\
\mathtt{or}\ \mathtt{msg_3} \Rightarrow \mathtt{a}(\mathtt{r_3}, \mathtt{C})
\end{array}
$$

$$
\begin{array}{l}
\mathtt{a}(\mathtt{r_3}(\mathbf{E}), \mathbf{C}) :: \\
\quad \mathtt{msg_3} \Leftarrow \mathtt{a}(\mathtt{r_1}, \mathtt{A}) \\
\quad \mathtt{then} \\
\quad \mathtt{a}(\mathtt{r_3}(\mathtt{En}), \mathtt{C}) \leftarrow \mathtt{En} = \mathtt{E} + 1
\end{array}
$$

Figure 6: LCC example 1 (top) and LCC example 2 (bottom)

The LCC example in the top section of Figure 6 defines the behaviour of a peer subscribing to a role $\mathtt{r}_1$. The peer, uniquely identified by variable A, first tries to satisfy the constraint `obtain(Data)`: if it succeeds, the variable `Data` is instantiated and sent within a message $\mathtt{msg}_1$ to peer `B` in role $\mathtt{r}_2$. It then waits for message $\mathtt{msg}_2$ from the same peer. If it fails to satisfy `obtain(Data)`, the message $\mathtt{msg}_3$ is sent to peer `C` in role $\mathtt{r}_3$.

The LCC example in the bottom section of Figure 6 defines the behaviour of a peer, identified in variable C, that takes role r_3: it waits for a message msg_3 from a peer in role r_1, and then recursively increments a counter.

In its definition, LCC makes no commitment to the method used to solve constraints - so different participants might operate different constraint solvers (including human intervention).

4. LCC EXAMPLE - FAN-IN

In order to demonstrate the feasibility of our service choreographic approach, we have implemented the fan-in pattern described in Section 2.1.1 using OpenKnowledge. The fan-in pattern LCC implementation is represented in Figure 8, along with the UML sequence diagram in Figure 7 of a run with three sources and one processor .

Peers agree to participate in interactions by subscribing to published interaction models. The action of subscription may be proactive, decided directly by a peer, or may be reactive, after an explicit request by another peer. In this specific interaction model, there can be any number of peers subscribed as `source`, and any number subscribed as `processor`, providing different types of data and different types of processing on the incoming data. During run-time, there still can be any number of `sources`, but there can be only one `processor`: the selection is performed during interaction bootstrap.

Assuming that the roles `processor` and `sources` are subscribed by the correct number of peers, when a peer subscribes to the role `user`, the DDS starts the interaction bootstrap process, forwarding all the subscriptions to a randomly chosen peer that acts as a coordinator. During bootstrap, the group of actual participants is selected; in this case the meaningful selection is likely performed by the `user` who specifies the data sources and the processor that are required. After this initial phase is complete, each participant is aware of every other participant taking part in the interaction.

Once the interaction starts, the peer in role `user` is asked by constraint `prepare_query(Q)` to define the query, that is forwarded to all the peers playing the role `sources`. Their identity is obtained by solving the built-in constraint `getPeers(SourceName, PeerList)`, that every peer can solve: the list of peers per role is bound during bootstrap, and is broadcast to all the participants. To send the query to all the recipients, the user changes role and assumes the subrole `querier`, that recurses over the list of sources `Ps` by sending the message `query(Q)` to the source contained in the head of the list until the list is empty.

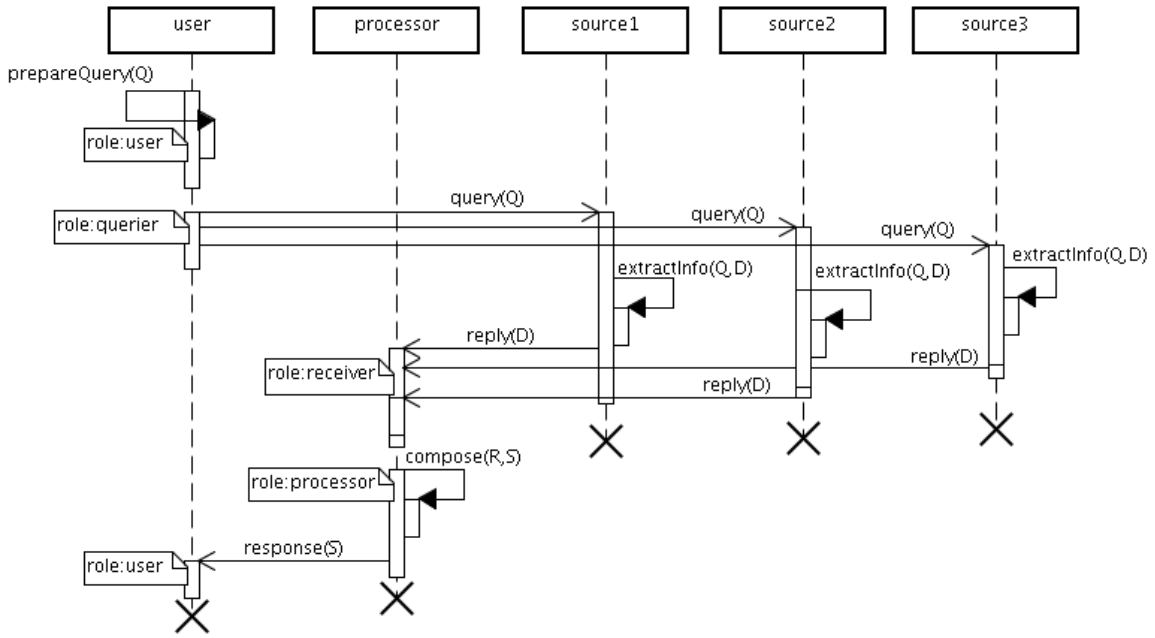


Figure 7: Sequence diagram for a run of the distributed fan-in interaction with 3 sources and 1 sink

---

$\mathbf{a(user, U)}$ ::
`null` $\leftarrow$ `prepare_query(Q)` and `getPeers("source",Ps)`
`then`
`a(querier(Q,Ps),U)` `then`
`response(S)` $\Leftarrow$ `a(processor,P)`

$\mathbf{a(querier(Q, Ps), U)}$ ::
`null` $\leftarrow$ `Ps` $= []$
or (`query(Q)` $\Rightarrow$ `a(source,P)` $\leftarrow$ `Ps` $=$ `[P|Pt]` `then` / `a(querier(Q,Pt),U)`)

$\mathbf{a(processor, P)}$ ::
`null` $\leftarrow$ `getPeers("source",Ps)` `then`
`a(receiver(NS,R),P)` $\leftarrow$ `len(Ps,NS)` `then`
`response(S)` $=>$ `a(user,U)` $\leftarrow$ `compose(R,S)`

$\mathbf{a(receiver(NS, R), P)}$ ::
`null` $\leftarrow$ `Ns` $==$ `0`
or (`reply(D)` $\Leftarrow$ `a(source,P)` `then` / `a(receiver(Rn,NSn),P)` $\leftarrow$ `Rn` $=$ `[D|R]` and `NSn` $=$ `NSn` $-$ `1`)

$\mathbf{a(source, S)}$ ::
`query(Q)` $\Leftarrow$ `a(user,U)` `then`
`reply(D)` $\Rightarrow$ `a(receiver,P)` $\leftarrow$ `extract_info(Q,D)`

Figure 8: LCC model for fan-in pattern

Each source receives the request message and processes it by solving the constraint `extract_info(Q,D)`. Once the information has been extracted, it sends the data directly to the `processor`. The processor is aware of the number of `sources`, having solved the constraints `getPeers("source",Ps)`, to obtain the list of participating sources and then the constraint `len(Ps,NS)` to obtain their number. In order to asynchronously receive each reply, the `processor` assumes the subrole `receiver`, that waits for message `reply(D)` to arrive from a `source` and then recurses, augmenting the list of received replies. When all the sources have sent their message, the processor returns to its main role, composes the different replies using some domain specific processing algorithm with constraint `compose(Rs,S)` and sends them directly to the `user` with message `response(S)`. There can be variations in the protocol: the `user` can send a message to the `processor` specifying various constraints, such as the maximum time to wait, or the minimum number of acceptable answers from the `sources`.

4.1 Expressiveness of LCC

We have demonstrated how it is possible to represent with a relatively simple protocol and implement with currently available technology a fully distributed approach for the fan-in pattern introduced in Section 2 . Fan-out and sequence patterns are similarly represented and executed.

Service Interaction Patterns [5], discussed in Section 2.1 facilitate the assessment of emerging Web services standards by providing a common basis on which workflow languages can be compared. LCC does not have dedicated constructs for dealing with these common interactions, i.e. no multicast support, no atomic transaction support. It is outside the scope of this paper to describe and implement each of the patterns, this is left to further work. Even with a lightweight syntax, LCC can implement the majority of patterns.

5. RELATED WORK

5.1 Choreography Languages

The *Web Services Choreography Description Language (WS-CDL)* is the proposed standard for service choreography. However, WS-CDL has met criticism [4, 10] through the Web services community. It is not within the scope of this paper to provide a detailed analysis of the constructs of WS-CDL, this research has already been presented [12]. However, it is useful to point out the key criticisms with the language: WS-CDL choreographies are tightly bound to specific WSDL interfaces, WS-CDL has no multi-party support, no formal foundation, no explicit graphical support and incomplete implementations.

Let's Dance [32] is a language that supports service interaction modelling both from a global and local viewpoint. In a global (or choreography) model, interactions are described from the viewpoint of an ideal observer who oversees all interactions between a set of services. Local models, on the other hand focus on the perspective of a particular service, capturing only those interactions that directly involve it.

BPEL4Chor [9] is a proposal for adding an additional layer to BPEL to shift its emphasis from an orchestration language to a complete choreography language. BPEL4Chor is a collection of three artifact types: participant behaviour descriptions, participant topology and participant groundings.

5.2 Discussion

There are an overwhelming number of pure orchestration languages. However, relatively few targeted specifically at choreography, in that sense LCC is a contribution in itself. There are even fewer complete implementations of choreography languages. However, in order to enact and validate service choreographies, we consider a concrete implementation a priority. LCC has a complete implementation through the OpenKnowledge framework which provides functionality to enact distributed choreographies over a peer-to-peer network. In comparison, there are only two documented, prototype implementations of the WS-CDL specification. WS-CDL+, an extended specification [15] has been implemented in prototype form, although only one version, version 0.1 has been released. A further partial implementation [12] of the WS-CDL specification is currently in the prototype phase. The other widely known implementation is `pi4soa` [24], an Eclipse plugin which provides a graphical editor to compose WS-CDL choreographies and generate from them compliant BPEL. Maestro [8] is an implementation of the Let's Dance language and supports the static analysis of global models, the generation of local models from global ones, and the interactive *simulation* (not enactment) of both local and global modes. In the BPEL4Chor space, A Web-based editor [7] allows engineers to graphically build choreography models.

In contrast with WS-CDL, whose original aim was to simply describe choreographies between collaborating services and lacked a formal semantics, and BPEL4Chor, whose semantics was defined through translation into extended Petri Nets [20], LCC semantics are derived directly from π-calculus. This has allowed, for example, to write a model-checker for LCC choreographies [23].

Furthermore, LCC is an executable choreography language. Peers do not have to be pre-configured with a particular choreography definition in advance, as described in the boot strapping process in Section 3.1. This is a more flexible, dynamic solution as definitions of choreography using WS-CDL, Let's Dance and BPEL4Chor are usually configured at design-time.

5.3 Techniques in Data Flow Optimisation

For completeness, it is important to consider architectures which have addressed the workflow bottleneck problem. The *Circulate* architecture [3] maintains the robustness and simplicity of centralised orchestration, but facilitates choreography by allowing services to exchange data directly with one another. Performance analysis [2] concludes that a substantial reduction in communication overhead results in a 2–4 fold performance benefit across all workflow patterns. End-to-end patterns demonstrate how the advantage of using the *Circulate* architecture increases as the complexity of a workflow grows.

Service Invocation Triggers [6] 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.

In [21] 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 Workflow Networks (EWFN), a Petri nets dialect.

Triana [28] 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. *OGSA-DAI [16]* middleware supports the exposure of data resources on to Grids and facilitates data streaming between local OGSA-DAI instances. *Grid Services Flow Language (GSFL) [19]* addresses some of the issues discussed in this paper in the context of Grid services, in particular services adopt a peer-to-peer data flow model. Finally, the Grid Superscalar [11] architecture is based on centralised control but facilitates nodes to store data where they are generated and forward them directly to where they are required.

6. CONCLUSIONS

This paper argues that as the number of services and the size of data involved in workflows increases, traditional centralised orchestration techniques are reaching the limits of scalability. Choreography techniques, although more complex to model, offer a decentralised alternative and for data-centric workflows, are the optimal architecture.

Through a number of workflow patterns, we have argued the case for choreography, focusing primarily on optimising data flow by removing the notion of intermediate data passing through a centralised orchestration engine.

While orchestration is the dominant architectural approach, there are relatively few pure choreography languages and even fewer concrete implementations. To add to the body of research, we have introduced the Light Weight Coordination Calculus (LCC), a type of process calculus used to formally define choreographies and its corresponding implementation, the OpenKnowledge frame-

work, a choreography-based architecture, providing the functionality for peers to coordinate in an open peer-to-peer system. The fan-in pattern was implemented to demonstrate both LCC and the OpenKnowledge framework. Extensive data flow optimisation literature was reviewed, covering pure choreography languages, decentralised orchestration techniques and Grid toolkits.

6.1 Future Work

The hand-encoding of protocols into the LCC formalism remains a time-consuming process. We are therefore currently considering a number of approaches which will permit protocols to be constructed in a more efficient manner. The simplest approach is the provision of a graphical tool for constructing protocols. Beyond this, we would like to support the automatic generation of protocols, i.e. the outcome of a planning process.

Full pattern analysis of LCC based on the Service Interaction patterns is currently underway.

7. REFERENCES

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

[2] A. Barker, J. B. Weissman, and J. van Hemert. 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, June 2008.

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

[4] A. Barros, M. Dumas, and P. Oaks. A Critical Overview of the Web Services Choreography Description Language (WS-CDL). BPTrends Newsletter 3, March 2005.

[5] A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns. In *Proceedings of the 3rd International Conference on Business Process Management*, pages 302–318. Springer Verlag, 2005.

[6] W. Binder, I. Constantinescu, and B. 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.

[7] BPEL4Chor Editor. `http://www.bpel4chor.org/editor/`.

[8] G. Decker, M. Kirov, J. M. Zaha, and M. Dumas. Maestro for Let's Dance: An Environment for Modeling Service Interactions. In *Demonstration Session of the 4th International Conference on Business Process Management (BPM), Vienna, Austria*, September 2006.

[9] G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proceedings of the IEEE 2007 International Conference on Web Services (ICWS 2007)*, pages 296–303, July 2007.

[10] G. Decker, H. Overdick, and J. M. Zaha. On the Suitability of WS-CDL for Choreography Modeling. In *Proceedings of Methoden, Konzepte und Technologien für die Entwicklung von dienstebasierten Informationssystemen (EMISA), LNI P-95, Hamburg, Germany*, pages 21–34, October 2006.

[11] V. Dialinos, R. M. Badia, R. Sirvent, J. M. Perez, and J. Labarta. Implementing phylogenetic inference with grid superscalar. In *The 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 1093–1100. IEEE Computer Society, 2005.

[12] L. Fredlund. Implementing WS-CDL. In *Proceedings of the second Spanish workshop on Web Technologies (JSWEB 2006)*, Universidade de Santiago de Compostela, November 2006.

[13] D. Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, January 1995.

[14] J. C. Jacob, D. 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.

[15] Z. Kang, H. Wang, and P. C. Hung. WS-CDL+: An Extended WS-CDL Execution Engine for Web Service Collaboration. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 928–935, 2007.

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

[17] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. Web Services Choreography Description Language (WS-CDL) Version 1.0. W3C Candidate Recommendation, November 2005.

[18] S. Kotoulas and R. Siebes. Deliverable 2.2: Adaptive routing in structured peer-to-peer overlays. Technical report, OpenKnowledge.

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

[20] N. Lohmann, O. Kopp, F. Leymann, and W. Reisig. Analyzing BPEL4Chor: Verification and participant synthesis. In M. Dumas and R. Heckel, editors, *Web Services and Formal Methods, Forth International Workshop, WS-FM 2007 Brisbane, Australia, September 28-29, 2007, Proceedings*, volume 4937 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2008.

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

[22] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:3045–3054, November 2004.

[23] N. Osman, D. Robertson, and C. Walton. Run-time

model checking of interaction and deontic models for multi-agent systems. In *Proceedings of the Third European Workshop on Multi-Agent Systems*, pages 248–259, Brussels, Belgium, December 2005.

[24] pi4soa. `http://sourceforge.net/projects/pi4soa`.

[25] D. Robertson, C. Walton, A. Barker, P. Besana, Y.-H. Chen-Burger, F. Hassan, D. Lambert, G. Li, J. McGinnis, N. Osman, A. Bundy, F. McNeill, F. van Harmelen, C. Sierra, and F. Giunchiglia. Interaction as a grounding for peer to peer knowledge sharing. In *Advances in Web Semantics*, volume 1. LNCS-IFIP, 2007.

[26] S. Ross-Talbot. Orchestration and Choreography: Standards, Tools and Technologies for Distributed Workflows. In *Workshop on Network Tools and Applications in Biology (NETTLAB 2005)*, October 2005.

[27] R. Siebes, D. Dupplaw, S. Kotoulas, A. P. de Pinninck, F. van Harmelen, and D. Robertson. The openknowledge system: an interaction-centered approach to knowledge sharing. In *Proceedings of the 15th Intl. Conference on Cooperative Information Systems (CoopIS)*, 2007.

[28] I. Taylor, M. Shields, I. Wang, and R. Philp. 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.

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

[30] The OpenKnowledge Framework. `http://www.openk.org`.

[31] W.M.P van der Aalst and A.H.M. ter Hofstede and B. Kiepuszewski and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3), 2003.

[32] J. M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede. Let's Dance: A Language for Service Behavior Modelling. In R. Meersman and T. Z, editors, *OTM Conferences (1)*, volume 4275 of *LNCS*, pages 145–162. Springer Verlag, 2006.