Decentralized and Reliable Orchestration of Open Services

Abul Ahsan Md Mahmudul Haque and Weihai Yu
University of Tromsø – The Arctic University of Norway
Email: {Mahmudul.Haque, Weihai.Yu}@uit.no

***Abstract*—An ever-increasing number of clouds and web applications are providing open services to a wide range of applications. Whilst traditional centralized approaches to services orchestration are successful for enterprise service-oriented systems, they are subject to serious limitations for orchestrating the wider range of open services. Decentralized orchestration provides an attractive option for applications based on open services. However, decentralized approaches are themselves faced with a number of challenges, including the possibility of loss of dynamic run-time states that are spread over the distributed environment. This paper presents a dynamic replication scheme for a decentralized approach to orchestration of open services, where a network of agents collectively orchestrate open services using continuation-passing messaging.**

Keywords-web service; peer-to-peer; replication.

I. INTRODUCTION

An increasing number of individuals and enterprises are having an increasing number of their data and business functionality on line and in the cloud. To enable new applications to access these data and functionality, cloud providers and online business applications are offering open services through published Application Program Interfaces (APIs). Service orchestration is the coordination and conduct of the execution of multiple open services in the new applications [1].

Two technologies are highly relevant to the support of applications built on top of open services. (1) Web mashups are web applications that use content from multiple open services. ProgrammableWeb (www.programmableweb.com), for instance, lists thousands of open services and mashups. Although web mashups have been around for several years, they are still very limited in functionality (i.e., content only) and systematic support. Most noticeably, they are typically hand-crafted with low-level programming details. Execution of external open services are conducted by the web servers running the mashups. (2) Service-oriented computing (SOC) has been very well developed and supports most of the functionality such open-service based applications need. Traditionally, SOC focuses on cost-effective construction and integration of sophisticated applications within and across organizational boundaries. Therefore, unlike applications based on external open services, services composed in SOC generally limit themselves within enterprises or between enterprises with mutual agreements (this is generally known as services choreography [1]). Usually, dedicated central engines carry out the orchestration of composite services.

Recently, there have been efforts that bring the SOC technology to the cloud and open-service based applications. For example, Amazon SWF [2] allows applications to coordinate work (including service invocations) across distributed components.

In all current approaches, services are orchestrated either by dedicated central engines (SOC), or by sites hosting applications (mashups). This clearly has advantages, such as control and overview of global run-time status. However, application sites are typically vulnerable with respect to availability, scalability and reliability, whereas finding feasible central engines is hard when the services are beyond enterprise boundaries [3]. Even if such an engine exists (as with Amazon SWF), relying on central engines and/or individual big-name vendors would be subject to issues like censorship, surveillance, policy-dependence etc. [4]. Furthermore, because open services are potentially spread all over the world, long network delays are unavoidable when the locations of central engines are fixed.

Based on the above observations, we believe a decentralized or peer-to-peer approach to open-services orchestration would be an attractive option to a wide range of next generation open-service based applications. There have been research activities in the SOC community on decentralised orchestration of services (more on these in Section VII on related work). It is generally challenging to support reliable orchestration of external services that could be unreliable. It is even more challenging for decentralized orchestration over a large group of unreliable peers or agents.

Our decentralized orchestration mechanism is called *continuation-passing messaging* (CPM) [5][6]. Our earlier work addressed issues with exception handling and recovery in order to support reliable orchestration when external services are unreliable. In this paper, we present a dynamic replication scheme for reliable orchestration with potentially unreliable orchestration agents.

The paper is organized as the following. Section II gives a motivating example. Section III presents a peer-to-peer system model for services orchestration. Section IV reviews CPM. Section V presents replicated CPM, the main contribution of this paper. Section VI presents performance results. Section VII discusses related work. Section VIII concludes.

II. EXAMPLE

Consider an application that assigns reviewers to papers submitted to a conference for reviewing. The application achieves this by doing the following. It first uses digital library L to get a ranked list of candidate reviewers for each submitted paper based on the title and keywords of the paper as well as the publications of the candidate reviewers. Then, for the candidates above a certain threshold, it uses citation indexing service I to refine the shorted list based on co-authorship, affiliation and citations. Finally, it uses the refined list and its on-premise data, such as reviewers' interests, to assign the reviewers to the paper.

 ISBN: 978-1-61208-337-7

The application may handle different exceptions differently. If during execution the digital library becomes unavailable, it may try another digital library. If the citation indexing services becomes unavailable, it may simply accept the ranking list generated so far without any further refinement. It may be important for the conference organizer that once the application starts to run, it keeps running until the final assignments are done.

When the services of the example application are orchestrated by a central engine or by the site running the application, for every paper-reviewer pair, there is at least a round trip of messages between the engine and the service. Ideally, the engine can be placed close to the open service, and the placement can be done at run time. This is the basic idea behind continuation-passing messaging.

III. System Model

A *service provider* (*SP*) provides services through an open API with a number of operations. We use S_a, S_b etc., to denote SPs and a, a_1, a_2, $\bar{a}$ etc., to denote operations provided by S_a. Operation a of S_a is invoked with message $invoke(a)$ to S_a. A *service-based application* (*SA*) consists of invocations to a number of service operations in a given prescribed order. Without loss of generality in our study, we adopt a service-composition model similar to Web Services Business Process Execution Language (WS-BPEL) [7].

Fig. 1 shows an example SA p that consists of invocations to operations a at S_a, b at S_b, c at S_c and d at S_d. The SA first invokes a and then forks two parallel branches. The first branch invokes b n times in a loop. The second branch invokes c and d in sequence.

p: $scope($
  $sequence($
    $invoke(S_a, a, \bar{a})$,
    $fork($
      $loop(n, invoke(S_b, b, \bar{b}))$,
      $scope($
        $sequence($
          $invoke(S_c, c)$,
          $invoke(S_d, d, \bar{d}))$,
        any: $sequenc(compensate, invoke(S_e, e))))))$,
  any: $compensate)$

Figure 1. An example SA.

We assume that operations a, b and d have reverse operations $\bar{a}$, $\bar{b}$ and $\bar{d}$, and that operation c is read-only and does not need a reverse operation. The element $invoke(S_a, a, \bar{a})$ means: "run service operation a at S_a; if p has to be rolled back due to an exception that occurs after operation a successfully returns but before the entire p finishes, run service operation $\bar{a}$ to compensate for the executed effect of a". Notice that $invoke(S_a, a, \bar{a})$ is an SA construct that is not understood by S_a. S_a only understands either $invoke(a)$ or $invoke(\bar{a})$.

A *scope* is a unit of exception handling. Exception handlers are associated with scopes. When an exception of certain type is thrown in a scope, all current activities in the scope are stopped and the corresponding exception handler is executed. In Fig. 1, the top level scope has an exception handler for *any* type of exceptions. It runs a single operation *compensate* that rolls back the current execution using the recovery plan that

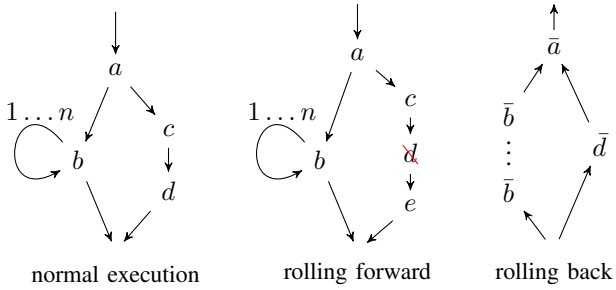


Figure 2. Control flow of example SA.

is automatically generated during the execution. The exception handler of the inner scope instead first rolls back the execution of the scope so far and then rolls forward by invoking an alternative service operation e.

Fig. 2 shows the control flows of a normal execution, a rolling forward (after the execution of d failed) and a rolling back (just before the entire p is about to finish).

Fig. 3 shows the service invocation messages (blue lines with arrows) when the SA is orchestrated by the site S_p that runs p. In this particular example, when the geographical distance between S_p and S_b is long, the loop may incur a long delay. If S_p is a mobile device, the execution of p could be costly and unreliable.

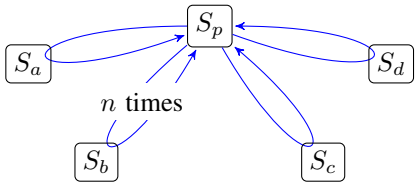


Figure 3. Centralized orchestration by the host SA server.

In our decentralized approach, a network of *orchestration agents* (OAs) collectively orchestrate the executions of SAs using *continuation-passing messaging* (CPM) [6]. We use A, A_a, A_b etc., to denote OAs. An OA has a *coverage* of SPs. Suppose a is an operation of S_a that is under the coverage of A_a. When a is part of an SA, invocation of a can be made via A_a.

At a specific moment, SPs may or may not be covered by OAs and OAs may have overlapping coverages. SPs become covered by OAs either by registration or through a learning process. An OA may not have the complete knowledge about the coverages of other OAs. At present, we assume that OAs learn effectively and every OA has nearly complete global knowledge of OA coverages.

An OA can run on a dedicated server, such as provided by a cloud provider. Alternatively, an SP may volunteer to be an OA as well. Being an OA may make its service more attractive. For example, if either S_b or the cloud hosting S_b has an OA, the loop in p may appear to be much more effective [5][6].

The basic tasks for the management of the OA network include OA membership, detection of OA availability, registration and discovery of SPs for their coverage, etc. Some of the tasks are already provided by existing software (such as the open source SERF [8]).

An SP may be unavailable, due to disconnection or system crash, and does not respond to invocations. An SP may also return an error. We assume that business critical services

 ISBN: 978-1-61208-337-7

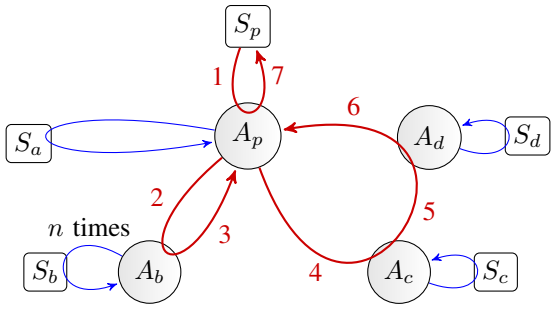
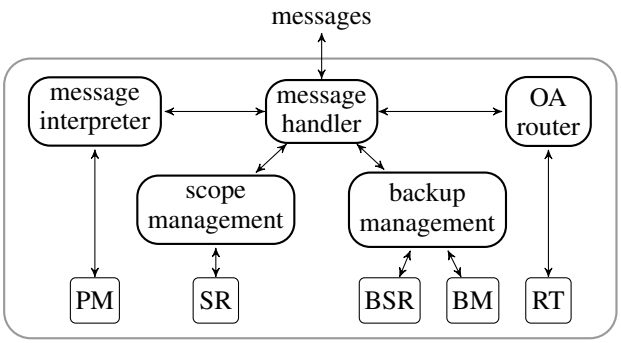


Figure 4. Messages with CPM Orchestration.



PM: pending messages, SR: scope registry, RT: routing table
BSR: backup scope registry, BM: Backup Messages

Figure 5. Structure of an Orchestration Agent.

support the at-most-once operation semantics. That is, an SP can recognize duplicated invocations and execute the same invocation at most once.

When an SP is not available or returns an error message, an exception is thrown so that an appropriate exception handler of the SA will handle it, such as by invoking an alternative service or rolling back the execution so far. Our orchestration mechanism guarantees effective propagation and handling of exceptions.

An OA may become unavailable in two ways. It may leave the OA network intentionally, or it may crash or get disconnected due to network failures. We assume a fail-stop crash model. The replicated CPM enhances the availability of the orchestration when the OAs are subject to such unavailability.

IV. CPM Overview

Fig. 4 shows the messages for CPM orchestration of the example SA p. Here we assume that SPs S_b, S_c and S_d are covered by OAs A_b, A_c and A_d, and S_a is not covered by any OA. There are three types of messages for services orchestration: service invocation messages (blue lines), CPM messages (red lines) and scope management messages (not shown in the figure). Orthogonal to the messages for services orchestration, OAs exchange routing messages to update the routing and health status of other OAs [6].

During orchestration, information like activity execution order and SA-aware data is carried in CPM messages in terms of continuations and environments. A *continuation* is a stack of activities that will be carried out, beginning from the head of the stack. An *environment* contains information of activity status and SA-aware data. To facilitate exception handling, messages also contain *compensation continuations*, which are rollback plans automatically generated during SA execution.

The initial continuation and environment of a CPM message are generated when an OA starts to orchestrate an SA. The message is later on sent to subsequent OAs that independently interpret the messages and invoke the service operations of the appropriate SPs. New continuations and environments are generated based on the messages being interpreted, as well as the outcomes of the service executions.

Fig. 5 shows the overall structure of an OA. In an OA, a *message interpreter* interprets an incoming or local message according to the head activity of the continuation. The following may happen during the interpretation.

An initial SA is converted into a CPM message. OAs are assigned to the corresponding activity elements according to the information in the OA router. For our example SA, $orch(p, S_p)$ —orchestration of p from S_p as specified in Fig. 1— is converted to $orch^{A_p}(scope^{A_p}(sequence(invoke^{A_p}(S_a, a, \bar{a})\ldots)))$, where orchestration activities like $orch$ and $scope$ are assigned to OAs A_p etc. For the purpose of space and readability, in what follows, we use notations like $scope^{A_p}(-)$ to suppress the details of the $scope$ element.

In some cases, a message can be interpreted alone. For example, $orch^{A_p}(scope^{A_p}(sequence(-))$ is interpreted into $scope^{A_p}(sequence(-)) \cdot eorch^{A_p}(-)$, which in turn is interpreted into $sequence(-) \cdot eos^{A_p}(-) \cdot eorch^{A_p}(-)$. Here $eorch$ and eos stand for *end-of-orchestration* and *end-of-scope*.

In other cases, multiple messages must be available to be further interpreted, for example, when messages from multiple parallel branches join. In this case, the first arrived messages are put in the *pool of pending messages* (PM). They are further interpreted when all dependent messages are available.

The interpretation of a message or multiple messages may lead to one or more new messages. Some messages are further interpreted locally by the same OA, like the $orch^{A_p}(-)$ above, and some are sent to other OAs for further interpretation.

If the head element of the continuation is an invocation assigned to the OA, the OA sends an invocation to the SP and waits for the result by putting a $wait$ message in its PM. For example, interpreting message $invoke^{A_p}(S_a, a, \bar{a}) \cdot fork(-) \cdot eos^{A_p}(-) \cdot orch^{A_p}(-)$, A_p sends $invoke(a)$ to S_a and puts $wait^{A_p}(S_a, a, \bar{a}) \cdot fork(-) \cdot eos^{A_p}(-) \cdot orch^{A_p}(-)$ in its PM. The $wait$ message is further interpreted according to the outcome of the invocation.

An OA may also be a scope manager and maintains some status information of each branch in its *scope registry* (SR). A scope manager is notified with a scope management message when the orchestration of an enclosing branch moves to a new OA. For example, when a branch moves from A_p to A_b, the scope manager A_p is notified of the move.

Table I lists the continuations in the remote CPM messages as shown in Fig. 4. Continuations of intermediate local messages are not shown in the table. In the table, κ is a continuation segment that is common in several continuations.

A $join$ element joins multiple parallel branches into one. It has an identifier and a join condition. Here the join condition is simply the number of branches to be joined.

The eos element marks the end of a scope and encapsulates necessary information for exception handling. The general form of an eos element is $eos^A(id, \kappa, h_1, h_2\ldots)$, where A

TABLE I. CONTINUATIONS IN MESSAGES

Msg	Continuation
1	$orch(scope(-), S_p)$
2	$invoke^{A_b}(S_b, b, \bar{b})$ $\cdot\, loop(n-1, invoke^{A_b}(S_b, b, \bar{b})) \cdot \kappa$
3	κ
4	$invoke^{A_c}(S_c, c) \cdot invoke^{A_d}(S_d, d, \bar{d})$ $\cdot\, eos^{A_p}(-) \cdot \kappa$
5	$invoke^{A_d}(S_d, d, \bar{d}) \cdot eos^{A_p}(-) \cdot \kappa$
6	$eos^{A_p}(-) \cdot \kappa$
7	$eorch(S_p)$
κ	$join^{A_p}(id_j, 2) \cdot eos^{A_p}(-) \cdot eorch^{A_p}(S_p)$

is the scope manager, id is the unique identifier of the scope, κ is a compensation continuation, and h_1, h_2 etc., are exception handlers. The compensation continuation is the recovery plan of the scope automatically generated during the orchestration. Table II lists the compensation continuations in the eos elements of the remote CPM messages. Notice that Messages 4, 5 and 6 have two eos elements for the two nested scopes.

TABLE II. COMPENSATION CONTINUATIONS

Msg	Compensation continuation	
1	nil	
2	$\bar{\kappa}$	
3	$invoke^{A_b}(S_b, \bar{b}) \cdots invoke^{A_b}(S_b, \bar{b}) \cdot \bar{\kappa}$	
4	$\bar{\kappa}$	nil
5	$\bar{\kappa}$	nil
6	$\bar{\kappa}$	$invoke^{A_d}(S_d, \bar{d})$
7	nil	
$\bar{\kappa}$	$join^{A_p}(id_{j'}, 2) \cdot invoke^{A_p}(S_a, \bar{a})$ $\cdot\, eos^{A_p}(-) \cdot eorch^{A_p}(S_p)$	

When A_b catches an exception, it runs the corresponding exception handler in the enclosing eos element and at the same time notifies the scope manager A_p of the exception. A_p then propagate the exception to the other branch(es).

For a scope with a single branch, an exception is completely handled where it is caught. This is the case of the nested scope of p. If A_d catches an exception, it handles the exception without notifying the scope manager A_p.

V. REPLICATED CPM

With CPM, information about the orchestration is usually already spread among multiple OAs. This information, if carefully maintained and updated, could be used to handle occasional unavailability of OAs. This is the key idea behind replicated CPM.

A. Selection of backup OAs

With replicated CPM, an SA orchestration has a *replication degree* k. That is, every activity is assigned with a list of $k+1$ OAs. The first OA in the list, called the *active* OA, is responsible for the interpretation of the message. The rest k OAs are *backup* OAs. For message c, we use $c.A$ for the active OA and $c.\mathcal{A}$ for the backup OAs. We also use $c.\mathcal{A}^+$ for the list of both c's active and backup OAs.

One of our primary goals for the selection of backup OAs is to reuse stored states and keep the run-time overhead of services orchestration as low as possible. The selection is based on the following observations:

- Every OA assigned with some activity for the orchestration will sooner or later obtain some information about the orchestration and this information would overlap with some backup information.
- To keep an OA updated with the information about an OA it backs up, it is often sufficient to send it the deltas of the latest changes, which are typically small fractions of the entire information.
- The amount of overlapping information, and therefore the sizes of the deltas, depends on the freshness of the currently stored information at OAs.

An important property of backup selection is that the backups of an OA can be unambiguously calculated by any OA at any time of the orchestration. This simplifies the handling of events like OA crashes.

The selection algorithm is built on *OA graphs* (OAG) of orchestrations. An OAG is first obtained with a projection of the control flow of the SA to the assigned OAs. If the number of OAs in an OAG is not sufficient for the number of backup candidates, it is extended with more OAs.

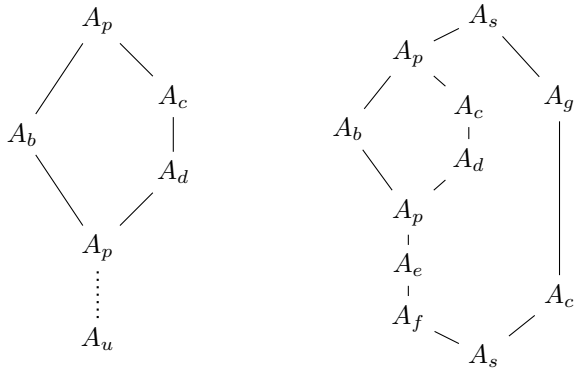
Fig. 6 shows the OAGs of an orchestration of the example SA p of Fig. 1 and an extension s with more OAs. In the OAG of p, A_p is a *parent* of A_b and A_c. If an OA is assigned to consecutive orchestration activities, the OA appears as a single node in the OAG. For example, if both $invoke(S_c, c)$ and $invoke(S_d, d, \bar{d})$ were assigned to A_d, only a single A_d node would have appeared in the OAG. On the other hand, the same OA may appear multiple times in an OAG if it is assigned to activities separated by other OAs. For example, there are two A_p nodes in the OAGs. Parallel branches are ordered. The ordering of branches are decided when an SA is initialized for orchestration. The general rule is that a branch with more orchestration activities has higher priority. For example, the branch with A_b has more orchestration activities than the other branch when n of the loop is larger than 2. In Fig. 6, a branch on the left has higher priority than a branch on the right.

The number of OAs in an OAG is the *degree* of the OAG. It determines the number of backup candidates each OA may have. If an orchestration of p requires that every OA should have 4 backup candidates, the minimum degree of the OAG is 5. This can be obtained by appending one more OA to the youngest node A_p, as A_u in Fig. 6. The selection of A_u is based on the information in the routing component, such as geographic distances.

The backup candidates of an OA A are selected with the following priority order:

S1. OAs of A's enclosing scopes have higher priorities than OAs of lower level nested scopes.
  a) Scopes closer to A have higher priorities.

S2. In a scope, OAs of the same branch have higher priorities than OAs of other branches.
In A's branch,
  a) Ascendant OAs have higher priorities than descendant OAs.



OAG of the example SA p OAG of an extended SA s

Figure 6. OA graph for backup selection.

b) OAs closer to A have higher priorities.

Among the other branches,

c) OAs of a higher-priority branch have higher priorities.

d) In the same branch, OAs closer to the scope manager have higher priorities.

Table III shows the lists of backup candidates (with length 4 for p and 7 for s) of the OAs for the two OAGs in Fig. 6.

TABLE III. BACKUP CANDIDATES

OA	p $(length = 4)$	s $(length = 7)$
A_s		A_p, A_e, A_f, A_g, A_c, A_b, A_d
A_p	A_b, A_c, A_d,A_u	A_s, A_e, A_f, A_b, A_c, A_d, A_g
A_b	A_p, A_c, A_d,A_u	A_p, A_c, A_d, A_s, A_e, A_f, A_g
A_c	A_p, A_d, A_b,A_u	A_p, A_d, A_b, A_s, A_e, A_f, A_g
A_d	A_c, A_p, A_b,A_u	A_c, A_p, A_b, A_s, A_e, A_f, A_g
A_e		A_p, A_s, A_f, A_g, A_c, A_b, A_d
A_f		A_e, A_p, A_s, A_g, A_c, A_b, A_d
A_g		A_s, A_c, A_p, A_e, A_f, A_b, A_d
A_c		A_g, A_s, A_p, A_e, A_f, A_b, A_d

The table only contains A_p and A_s once for each SA. The reason is that the backups for A_p is computed when A_p becomes a scope manager and it stays active until the end of the scope.

As an example, the backups for A_e, are selected according to the following rules of the selection algorithm: A_p (S1.a, S2.a, S2.b), A_s (S1.a, S2.a), A_f (S1.a, S2), A_g (S1.a, S2.c), A_c (S1.a), A_b (S2.c) and A_d.

During an orchestration, $c.\mathcal{A}^+$, the actual active and backup OAs for message c are selected from the first $k+1$ available OAs in the candidates OAs obtained from the OAG.

B. Normal execution

Every CPM message contains an integer k as the replication degree of the current branch, an OAG of degree l ($l > k$) and a list of actual active OA and backups.

In addition, every message has a *timestamp* that can be used to check causal relations between messages. A timestamp is of the form $[b_0, n_0] \cdot [b_1, n_1] \cdot \ldots$, where $b_0, b_1, \ldots$ are the unique identifies of the branches which the message is part of, and $n_0, n_1, \ldots$ are the sequence numbers in the branches. As shown in Fig. 7, in the beginning, there is only one branch (0). After a $fork$, two new branches $(0,0)$ and $(0,1)$ are created. The $orch$ message has sequence number 0 in branch (0). All

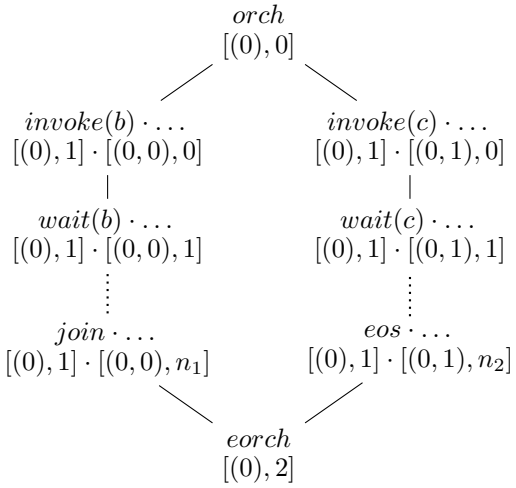


Figure 7. Message timestamps.

messages in the new branches have the same sequence number 1 in the parent branch (0), but different sequence numbers 0, 1, ..., in the new child branches (0.0) and $(0,1)$.

To compare the causality of two messages m^1 and m^2, we first get the longest prefix of their timestamps such that $b_0^1 = b_0^2, \ldots, b_i^1 = b_i^2$ $(i \geq 0)$. Message m^1 *happens before* Message m^2 in the same SA execution, denoted $m^1 \prec m^2$ or $m^2 \succ m^1$, if $n_0^1 = n_0^2, \ldots, n_{i-1}^1 = n_{i-1}^2$ and $n_i^1 < n_i^2$. Messages m^1 and m^2 are *concurrent*, denoted $m^1 \parallel m^2$, if $n_0^1 = n_0^2, \ldots, n_i^1 = n_i^2$.

Suppose that OA A, after interpreting a remote message c_0 and some local interpretations, is currently interpreting message c. Suppose further that the current scope manager and its backups are $c.S$ and $c.\mathcal{S}$ (and $c.\mathcal{S}^+ = \{c.S\} \cup c.\mathcal{S}$). The following are the steps related with sending messages during the orchestration of a normal execution:

C1. When the orchestration of a branch is moving away from A with CPM message c:
   a) Select $c.\mathcal{A}^+$.
   b) Send to $c.\mathcal{A}^+$ message c (or its delta).
   c) Notify $c.\mathcal{S}^+ \cup c_0.\mathcal{A} - c.\mathcal{A}^+$ about the move with message m. m contains two sets of OAs $c.\mathcal{S}^+$ and $\mathcal{G} = c_0.\mathcal{A} - c.\mathcal{A}^+$.

C2. When A stores a local message c in its PM, it also sends the delta of the message c_A to $c_0.\mathcal{A}$.

Step C1.a selects the next active OA and its backups according to the availability of OAs obtained from its RT. Step C1.b extends the destination of a CPM message to include the backups. Step C1.c has two purposes: 1) it extends a scope message to include the scope manager's backups ($c.\mathcal{S}^+$); 2) it informs some of A's backups ($\mathcal{G}$, which no longer backup the subsequent states of the same SA) to purge the backup states. Step C2 informs A's backups about its own state changes.

$c.\mathcal{S}^+$ in step C1.c was selected when the corresponding $scope$ element was interpreted. Step C1.c does not check the availability of the scope manager like step C1.a. The unavailability of an OA that has been active, like a scope manager, is handled in Subsection V-C.

Some messaging overhead is reduced when OAs play multiple roles. For example, when $c.\mathcal{A} = \{A\}$, which is typically true for $k = 1$ (according to the backup selection rules), step

C1.b does not involve any additional remote message than a non-replicated orchestration.

When OA A_r receives a CPM message c (or delta), it does the following:

R1. Ignore c if A_r has already received a message c' such that $c' \succ c$.
R2. If $A_r = c.A$, interpret c.
R3. If $A_r \in c.\mathcal{A}$, store or update backup status of $c.A$.
R4. If $A_r = c.S$, update scope state in SR.
R5. If $A_r \in c.\mathcal{S}$, update BSR.

If a message of a later stage of the same SA execution has already been processed, the newly arrived message is ignored (step R1). The message is handled depending on whether the receiver is an active OA (step R2), a backup OA (step R3), an active scope manager (step R3) or a backup scope manager (step R4).

When OA A_r receives a message m, notifying that an orchestration is moving from A to A', it does the following:

M1. Ignore m if A_r has already received a scope message m' such that $m' \succ m$.
M2. If $A_r = m.S$, update the status of scope in the SR.
M3. If $A_r \in m.\mathcal{S}$, update the backup status of the scope in BSR.
M4. If $A_r \in m.\mathcal{G}$, purge backup status of A.

Notice that in some situations, $c.\mathcal{A}^+ \cap c.\mathcal{S}^+ \neq \emptyset$, the tasks for steps M2 and M3 are done in R4 and R5. In general, the more these sets overlap, the more overhead is avoided.

C. Handling unavailability of OAs

When an OA becomes unavailable, its tasks for services orchestration, either as an active or backup OA, are taken over by other OAs. There are two types of tasks: interpretation of CPM messages and management of scopes. In this subsection, we focus on the first type, i.e., to continue interpreting CPM messages when an OA becomes unavailable. The steps to continue scope management is almost the same.

The unavailability of OAs is handled on a per-message basis, or a per-branch basis, because every CPM message represents an SA branch. When an OA in $c.\mathcal{A}^+$ becomes unavailable, it is always the highest ranked available OA in $c.\mathcal{A}^+$ to take the responsibility of handling the unavailability.

An OA becomes unavailable either when it leaves the OA network on purpose, or when it crashes or is disconnected due to some network failure. Before OA A leaves on purpose, it notifies the highest ranked available OA in $c.\mathcal{A}^+ - \{A\}$ for every message c in its PM and BM about its leaving. An OA A_r does the following when receiving this message:

L1. If A is the highest ranked OA in $c.\mathcal{A}^+$, A_r takes over as the actual active OA of c.
L2. Add a new OA to $c.\mathcal{A}^+$ according to the OAG and inform the new $c.\mathcal{A}^+$ about the latest update of c.

When an OA crashes or is disconnected from the network, its unavailability is detected when another OA is unable to send it a message. Because the OAs exchange routing messages regularly [6], the unavailability is detected in short time. Generally, the busier the OA network, the shorter the detection time. As soon as an unavailability is detected, it is propagated to the entire OA network.

When an OA A_r is notified of the unavailability of A, it finds relevant CPM messages in its PM and BM. A message c is relevant if $A \in c.\mathcal{A}^+$. For each such message c, it does the following:

U1. If A_r is the highest ranked available OA in $c.\mathcal{A}^+ - \{A\}$, do L1 and L2.

With respect to correctness, think of a message as representing a particular step of a branch. Because only the highest ranked available backup OA takes over the role as the new active OA of a message when the current active OA becomes unavailable (and once an OA is detected as unavailable, it will not be re-assigned to the same process execution when it becomes available again), it is impossible for two OAs to simultaneously take over as the new active OA of the same message.

However, backups of different messages of the same branch may coexist in different OAs. Consequently, different OAs may independently take over the role as the active OAs of different steps of the same branch. This does no harm when business critical services enforce the at-most-once execution model. In addition, if a scope manager observes that two OAs are responsible for the orchestration of the same branch, it kills the activities represented by the outdated messages. Eventually, the active OA of the most up-to-date message wins as the only active OA of the branch.

To make the last point clearer, consider this particular situation: OA A becomes unavailable just after an orchestration moved to the next OA A', and the notification of the unavailability arrives to A_r before the notification of the move (steps C1.b and C1.c in Subsection V-B). In this situation, A_r may take over and repeat the work that A had just finished before it became unavailable. The repeated work will eventually arrive at A'. By checking the timestamp of the message (step R1 in Subsection V-B), A' can figure out that the orchestration of this branch has already passed over this stage. The same is also detected by the scope manager (step M1). In the worst case, if a service invocation is repeated, a business-critical service will return with an exception due to the at-most-once semantics.

The last issue will not occur for replicated scope managers, because a scope manager never moves from OA to OA in the basic CPM scheme.

At this point, it should be clear that the replication scheme can tolerate up to k crashes during the time interval between the detection and the handling of an unavailability.

VI. Performance

We developed an OA prototype that runs in a simulator [9] for performance study. We study the performance of OAs with different degrees of replication and at different workload.

In our experiment, there are 100 SPs, 10 of the which are OAs as well. That is, these 10 sites both process service invocations and contribute to orchestration of services. Every OA covers 10 SPs. The distances between an OA and the SPs it covers are relatively short. An SP spends on average 100ms to process a service invocation. An OA spends on average 10ms to interpret a CPM message, and 1ms to handle a scope message, backup message or purge message. We model the workload with *multiprogramming levels* (MPLs) of SPs, which is the number of concurrent service operations it executes most of the time. Initially, a fixed number of SA executions are fed into the system. A new SA execution starts as soon as an existing one terminates. An SA execution consists of 4 operation invocations to different randomly chosen SPs.

 ISBN: 978-1-61208-337-7

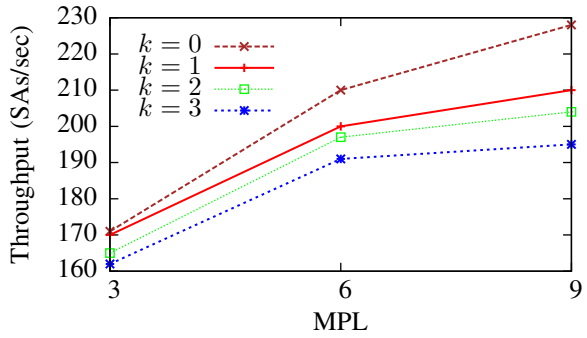


Figure 8. Throughput of 100 SPs.

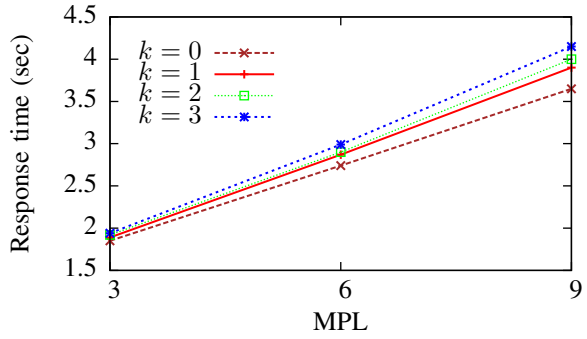


Figure 9. Response time of SAs.

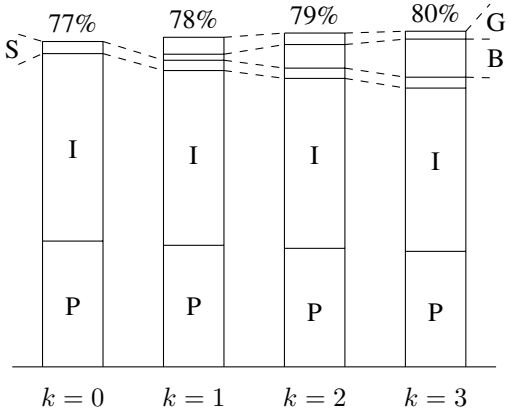
Fig. 8 shows the aggregate throughput of all SPs (measured in the number of completed SA executions per second). Fig. 9 shows the average response time of the SAs.

Fig. 10 shows the average resource utilization at OAs when the SP MPL is 6. We only show the resource utilization at one MPL, because although the total resource utilization varies at different MPLs, the proportion of different kinds of message handling is almost the same through all MPLs. As the degree of replication increases, the overhead of backup management ("B" and "G") increases, and the capability of normal service orchestration ("I" and "S") and service operation execution ("P") decreases. Consequently, the overall SP throughput decreases and SA response time increases, as shown in Fig. 8 and Fig. 9.

It is interesting to notice that when $k = 1$, the overhead of backing up orchestration states ("B") is less than the overhead of purging the backup states ("G"). The reason for this is that when an OA A forwards the orchestration to the next OA A', $c.\mathcal{A}^+ = \{A, A'\}$ in step C1 of Subsection V-B. In other words, A already has the state locally and the overhead of backing up the state is therefore low.

It is also interesting to notice that when k increases, the overhead of purging the backup states ("G") decreases. This is because an OA backups up the states of several stages of the same orchestration. When it store the backup state of a new stage, it also purges the state of an earlier stage. In other words, the larger overlap of $c_0.\mathcal{A}$ and $c.\mathcal{A}^+$ in step C1.c of Subsection V-B leads to the decrease of "G".

When an OA becomes unavailable, other OAs will handle the unavailability. We expected that this will cause a sudden increase of workload which will influence the overall performance of the system. For example, when k is 2 and SP MPL is 6, an OA covering 10 SPs is handling (most of the



P: service process, I: message interpretation, S: scope management
B: store/update of backup states, G: purge of backup states

Figure 10. Resource utilization at OAs at MPL 6.

time) 60 CPM messages and backing up 120 for other OAs. If an OA crashes, 180 messages will be handled by other OAs. However, in our experiments, we could not observe significant overall performance hiccup. The main observable difference in overall performance is that MPL of OAs has increased nearly 10%, both during handling of the unavailability and afterwards. It turns out that the messages that the unavailable OA was actively orchestrating (60 in this example) were the primary contributor to the increase of load at other OAs. The backup messages (120 in this example) contributed only very little to the increase of load at other OAs. More precisely, it is primarily the "I" part in Fig. 10 that contributed to the increase of load at the remaining OAs.

VII. Related Work

Decentralized orchestration in SOC research can be categorized into instantiation-based or messaging-based [5]. An instantiation-based approach [10][11][12][13][14] instantiates in advance a composition with resource and control allocation in the distributed environment. The allocated resources and control are responsible for the orchestration of the subsequent executions of the same composition. This approach is therefore more suitable for enterprise applications where allocation of resources is possible, and compositions are stable and are typically repeated many times [5]. In messaging-based approaches [5][10][15][16], information like execution order of activities is carried in messages. Since no resource or control is allocated in the distributed environment before an execution starts, messaging-based approaches would be more appropriate for orchestration where either the compositions or the environment are so dynamic that pre-allocation of resources is impractical.

The focus of research on reliable services orchestration has been on dealing with failures of constituent services, mostly based on compensation-based recovery [5][12][13][16]. Little work is done on dealing with failures of orchestration engines or agents.

Several replication schemes have been proposed in the research area of data streams and continuous queries. Gedik and Liu [17] applied a passive or backup replication mechanism to executions of continuous queries. A continuous query is executed on peers with matching ids. The selection of replicas

 ISBN: 978-1-61208-337-7

or backups is based on peer ids and neighbor proximity of the peer-to-peer network. In [18], data flow from sensors to data processing programs through an overlay network of peers. Peers are grouped into cells. Active replication is applied to the peers in the same cells to enhance the availability of the data flows. Martin, Fetzer and Brito [19] proposed an active replication scheme to a stream variant of map-reduce system consisting of stages of map-reduce operators. Replication is applied among data partitions of the same stage. The focus is on utilizing unused CPU cycles for replication. Zhang et al. [20] introduced a hybrid active/passive replication scheme to a peer-to-peer stream processing system to deal with transient failures due to high workload. It dynamically switches between active and passive schemes according to the workload in order to utilize the best part of both schemes.

The key difference of the afore-mentioned replication work and ours is that in continuous queries or data stream processing, tasks assigned to processing agents or peers are long lasting. It is therefore more suitable to have a relatively stable set of replicas and even special-purpose multicast communications among them.

Continuation-passing messaging was presented in more detail in our early work [5]. This early approach was however too intrusive. It requires that service providers support message interpretation. Although this might be arguably acceptable for enterprise applications, it is too strong a requirement for open services. Organization of OA networks for orchestration was later presented in [6]. Support for exception handling and rollback due to service failures was also presented in more detail in [5].

VIII. Conclusion

We presented a replication scheme for decentralized orchestration of open services with continuation-passing messaging. The scheme utilizes the knowledge about the control structure that is encapsulated in messages and the run-time state that is already spread in the distributed environment to enhance the availability of the orchestration. For a degree-k replicated orchestration, every branch can tolerate simultaneous crashes of up to k orchestration agents. Our performance study shows the overhead of replication during normal services orchestration.

There are still a number of issues to be addressed before the new approach can be practically adopted.

Security is always an important concern of distributed applications. We have not worked on security issues yet, but our approach is already useful when used in special cases. For example, if the orchestration agents are deployed at geographically different places by the same organization or a set of trusted applications, these agents can be used as a smart pool of orchestration engines where the orchestration activities are dispatched to the most appropriate engines.

The performance study shows that replication does incur a performance penalty. An incentive model would encourage more service providers to offer as orchestration agents. For example, applications that offer orchestration capabilities should have higher priority when scheduled and should be more entitled to higher degree of replication.

References

[1] C. Peltz, "Web services orchestration and choreography," Computer, vol. 36, no. 10, 2003, pp. 46–52.

[2] Amazon SWF: The Amazon Simple Workflow Service, [retrieved: April, 2014] http://aws.amazon.com/documentation/swf/.

[3] M. Wieland, K. Görlach, D. Schumm, and F. Leymann, "Towards reference passing in web service and workflow-based applications," in Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference (EDOC), Auckland, New Zealand, 2009, pp. 109–118.

[4] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta, "PeerSoN: P2P social networking: early experiences and insights," in Proceedings of the Second ACM EuroSys Workshop on Social Network Systems (SNS), Nuremberg, Germany, 2009, pp. 46–52.

[5] W. Yu and A. A. M. M. Haque, "Decentralised web-services orchestration with continuation-passing messaging," IJWGS, vol. 7, no. 3, 2011, pp. 304–330.

[6] A. A. M. M. Haque, W. Yu, A. Andersen, and R. Karlsen, "Peer-to-peer orchestration of web mashups," in The 14th International Conference on Information Integration and Web-based Applications and Services (iiWAS), Bali, Indonesia. ACM, 2012, pp. 294–298.

[7] Web Services Business Process Execution Language (WS-BPEL) Version 2.0, Organization for the Advancement of Structured Information Standards (OASIS), April 2007, [retrieved: April, 2014] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[8] SERF, [retrieved: April, 2014] http://www.serfdom.io/.

[9] A. Varga, "OMNeT++," in Modeling and Tools for Network Simulation, K. Wehrle, M. Günes, and J. Gross, Eds. Springer, 2010, pp. 35–59.

[10] A. Barker and R. Buyya, "Decentralised orchestration of service-oriented scientific workflows," in Proceedings of the 1st International Conference on Cloud Computing and Services Science (CLOSER), Noordwijkerhout, Netherlands, 2011, pp. 222–231.

[11] B. Benatallah, M. Dumas, and Q. Z. Sheng, "Facilitating the rapid development and scalable orchestration of composite web services," Distributed and Parallel Databases, vol. 17, no. 1, 2005, pp. 5–37.

[12] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda, "Decentralized orchestration of composite web services," in Proceedings of the 13th international conference on World Wide Web (WWW), New York, USA, 2004, pp. 134–143.

[13] G. J. Fakas and B. Karakostas, "A peer to peer (P2P) architecture for dynamic workflow management," Information & SW Technology, vol. 46, no. 6, 2004, pp. 423–431.

[14] P. Muth, D. Wodtke, J. Weißenfels, A. K. Dittrich, and G. Weikum, "From centralized workflow specification to distributed workflow execution," J. Intell. Inf. Syst., vol. 10, no. 2, 1998, pp. 159–184.

[15] D. Martin, D. Wutke, and F. Leymann, "A novel approach to decentralized workflow enactment," in Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC), Munich, Germany, 2008, pp. 127–136.

[16] T. Möller and H. Schuldt, "A platform to support decentralized and dynamically distributed P2P composite OWL-S service execution," in Proceedings of the 2nd Workshop on Middleware for Service Oriented Computing (MW4SOC), Newport Beach, California, USA, 2007, pp. 24–29.

[17] B. Gedik and L. Liu, "A scalable peer-to-peer architecture for distributed information monitoring applications," IEEE Transactions on Computers, vol. 54, no. 6, 2005, pp. 767–782.

[18] R. Martins, P. Narasimhan, L. Lopes, and F. Silva, "Lightweight fault-tolerance for peer-to-peer middleware," in Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS), New Delhi, India, 2010, pp. 313–317.

[19] A. Martin, C. Fetzer, and A. Brito, "Active replication at (almost) no cost," in Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS), Madrid, Spain, 2011, pp. 21–30.

[20] Z. Zhang et al., "A hybrid approach to high availability in stream processing systems," in Proceedings of IEEE 30th International Conference on Distributed Computing Systems (ICDCS), Genova, Italy, 2010, pp. 138–148.

 ISBN: 978-1-61208-337-7