UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

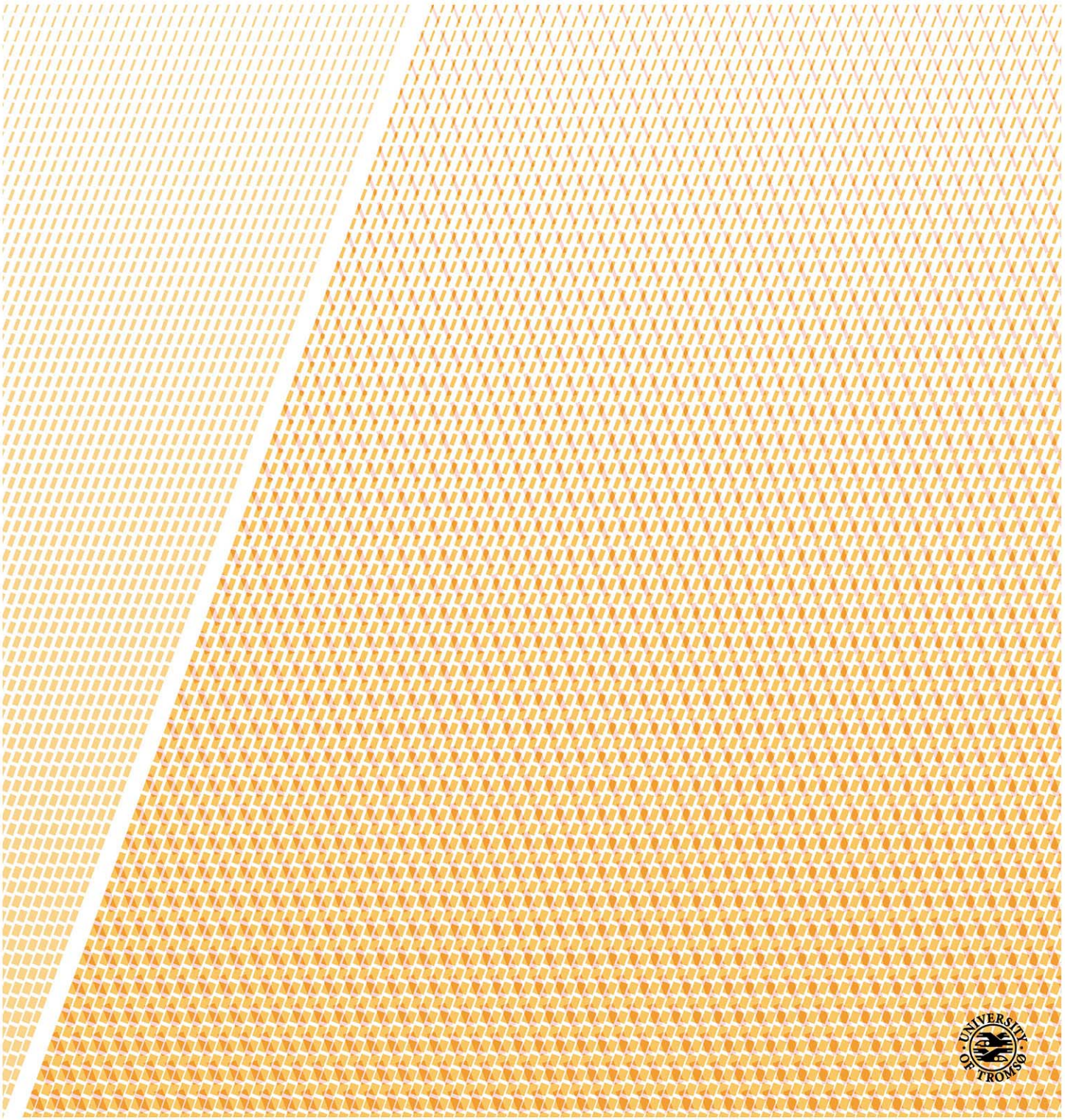
FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Decentralized Orchestration of Open Services

Achieving High Scalability and Reliability with Continuation-Passing Messaging

Abul Ahsan Md Mahmudul Haque

A dissertation for the degree of Philosophiae Doctor – August 2017



Acknowledgements

First and foremost I would like to express my deepest and sincere gratitude to my advisor, associate professor Weihai Yu. During this whole journey, I remain indebted for him not only for his continuous support for my Ph.D study and related research but also for his understanding and support during the times when I was really down. I am really thankful to him for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined completing this thesis without his continuous guidance and effective suggestions.

I would also like to thank my co-advisor associate professor Anders Andersen for his guidance and valuable comments during discussing experimental results and writing papers. I am also grateful to him for paying detail attention to my thesis and making helpful comments and valuable suggestions.

I would also like to thank other research fellows (specially Nazeeb and Dr. Razib Hayat) for their valuable discussions and giving me this believe that I can pursue and fulfill my research goals. I would also like to thank my current colleagues for their encouragements and supports which allow me to complete this thesis besides my current work.

Last but not the least, I would like to thank my family: my wife Sadia, my parents and my brother for supporting me spiritually throughout writing this thesis and my life in general.

Abstract

Orchestration of the executions of composite services in a service-oriented architecture is typically carried out by dedicated central engines. With central engines, monitoring and management of executions of composite services are relatively straightforward. However, a central engine can easily become a performance bottleneck when the number of services to be orchestrated is getting large. Furthermore, finding feasible locations for central engines is much harder when the services are beyond enterprise boundaries, especially for *open services*, services publicly available for wide range of applications, where naturally those services remain outside of the administration boundary of enterprises.

We investigate a *decentralized approach* as an alternative to centralized service orchestration. Decentralized orchestration, however, is generally regarded as more challenging for certain orchestration management tasks due to the absence of global run-time states. The hypothesis of this thesis is that *if we let the messages for service orchestration carry the control and status information about service executions, we could dispense with dedicated central engines.* Furthermore, *if we effectively utilize the current run-time states and future orchestration plans in the messages, we could eventually enhance the reliability of the executions of the composite services.*

The primary contribution of this dissertation is a fully decentralized approach to orchestration of open services. The approach is called *continuation-passing messaging* (CPM), where control and run-time state information are carried in messages in terms of continuations. Service orchestration is a process of exchanging and interpreting CMP messages. Our orchestration approach deviates considerably from other decentralized approaches as it does not require pre-allocation of resources to follow up the monitoring and management tasks.

Another important contribution of this dissertation is reliability of service orchestration. In our system model, failures may occur in one of two places: either at the service providers or at the orchestration

agents. We handle the first type of failures through exception handling of composite services and the second type with replication.

Exceptions of composite services could be handled either by backward recovery or forward recovery. The recovery plans can either be specified manually or generated automatically according to certain pre-defined rules. With CPM, we could automatically generate recovery plans at run time and encapsulate them in messages in terms of *compensation continuations*. We also devised a mechanism for monitoring the executions of services and propagating exceptions through scope managers.

We designed a special replication scheme called *replicated CPM*. It utilizes the run-time status information, which is already distributed among the participant orchestration agents for orchestration, and enhances the handling of the information for backup and replication purposes. It is a flow-oriented replication mechanism where failure of the orchestration agents is handled by the set of the backup agents that are chosen according to the structure of the composition. With replicated CPM, an orchestration activity has a *replication degree* k, meaning that, it is assigned with a list of $k + 1$ orchestration agents and can tolerate up to k simultaneous agent crashes.

Our performance study showed that decentralized orchestration improves the scalability of the orchestration process. Our orchestration approach has a clear performance advantage over traditional centralized orchestration and over the current practice of web mashups where application servers themselves conduct the execution of the composition of open web services. Finally, in our performance study we presented the overhead of the replication approach for services orchestration.

Contents

List of Figures

Chapter 1

Introduction

Service orientation [21] is a design paradigm for cost-effective construction and integration of sophisticated enterprise applications. This new genre of software paradigm finds its origin in object-oriented and component-based software development, and aims at enabling developers to build networks of interoperable and collaborative applications. Application developers could make use of independent computational units, primarily known as services, regardless of the platform where the applications and services run and of the programming language used to develop them [74]. Individually shaped services are composed to be collectively and repeatedly utilized to meet specific business goals. Traditional business process and workflow technologies have been successfully applied to service-oriented architectures for the orchestration of the composite services.

The World Wide Web [9], or simply the web, initially thought of as primarily for human use, has evolved towards an *Internet-Scale* application model that supports automated and repeated use of applications. Web applications targeted towards other applications are generally known as web services. The web service technology takes leverage from existing Internet technologies and related standards, and at the same time brings about new challenges.

1.1 Web services as open services

The web was initially designed primarily for human use. Lately, an ever-growing large number of web applications provide open services through published APIs (Application Programming Interfaces). New applications are built as the composition of the functionality and data from these open web services. A particular group of such open-service based applications, which also take the form of web applications, are widely known as web mashups. A *web mashup* is a web appli-

cation that uses other open web services. ProgrammableWeb[1], for instance, lists thousands of open services and mashup applications. According to their research center, the number of web APIs has increased thousand times from early 2005 til the end of 2013. Although open services show highest popularity among social media based applications, their popularity also ranges from sectors like finance, enterprise, mapping, e-commerce, etc. The APIs could be SOAP based (Simple Object Access Protocol) [79], RESTful (Representational State Transfer) [25], JSON, or combinations of SOAP and REST.

Currently, a web mashup can invoke individual open web services, but there is no systematic way of composing open web services as in service-oriented architectures. In this thesis we focus on and experiment with the run-time support of composite open services. We carefully manage the control information, run-time status and states of the open services, and further enact this information during the orchestration of the services.

1.2 Service composition and service orchestration

Available services, when work individually, may not fulfill the required functionality. For example, a map service and a bus routing service independently may not provide sufficient services for map-enhanced travel planning. However, when they work as a coordinated composition, they can perform to achieve the predefined goals.

Service orchestration is the process of conducting the coordinated executions of composite services. Web services and specially composite web services need to be orchestrated on the Internet, meaning that when a number of individual web services are glued together, we need to manage and monitor data and control flows of the composite service.

Services orchestration can be carried out either centralized or decentralized. In a centralized approach, orchestration of composite services is carried out by dedicated central engines. With central engines, monitoring and management of service executions are relatively straightforward. A central engine, however, can easily become a performance bottleneck when the number of services to be orchestrated is getting large. Furthermore, finding feasible locations for central engines is much harder for open web services which are typically beyond enterprise boundaries.

In contrast, a decentralized approach does not require a dedicated central engine. Instead, participant service providers or intermediate agents collabo-

[1] www.programmableweb.com

rate with each other and exchange or distribute messages over the network. As the overall control of the execution is distributed among the participating service providers or intermediate agents, there is no single point of performance bottleneck. On the other hand, decentralized approached are subject to other challenges, such as monitoring and management of run-time states, handling of failures, etc. We work on a new decentralized approach and deal with these particular challenges.

1.3 An evolution of web technology and open services

Here we present our perspective of the evolution of the web technology and open services, and how this thesis work fits in this evolution. Figure 1.1 depicts the evolution of web technologies starting from very generic client-server technique towards advanced service compositions. The numbers labeling the edges represent the possible technique for building web applications.

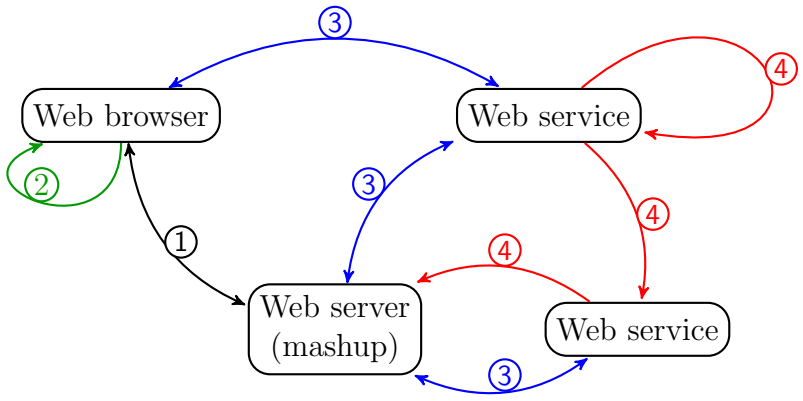


Figure 1.1: A perspective of the evolution of web technology

When the web first appeared in the late eighties, it was primarily used by human users. A human user interacts with a web browser, which obtains data content directly from web servers (marked as 1 in the figure).

With the advent of scripting languages inside browsers, such as Javascript, the browser can perform certain tasks, like validation of user input, without contacting the web server (marked as 2 in the figure).

When a web server provides open APIs as services, they can be used by programs, either from web browsers or from other web applications known as

web mashups (marked as 3 in the figure).

As a web mashup is essentially a composition of existing web services, a lot can be learned from service-oriented computing (SOC). Suppose that we would like to build a web application for a conference that reuses existing web services. The application offers useful information for the conference attendees, such as literature references and sight-seeing attractions, which it obtains from external services like digital libraries and tourist offices. The application can even automatically build interest groups and set up extra discussion sessions or social events using external social network services. It may happen that due to unexpected reasons, certain partially executed or completed operations need to be rolled back. For example, due to time conflicts or unavailability of conference rooms, establishment of some groups or sessions must be undone. Currently, no mashup building tool is able to support all features this application needs, such as exception handling and rollback. Most of these features are already well supported in different SOC approaches.

SOC focuses on cost-effective construction and integration of sophisticated applications within and across organizational boundaries. Therefore unlike web mashups, service compositions generally limit themselves within enterprises or between enterprises with mutual agreements. SOC is typically built on business process or workflow technology and provides more structural and feature-complete support than web mashups. [83] discusses the differences between mashup and SOC, with an emphasis on support for mashup application development.

Usually in SOC, dedicated central engines carry out the orchestration of composite services. However, finding feasible locations for central engines is hard when the services are beyond enterprise boundaries [81]. As mashup applications are by nature composed of services from different service providers, finding feasible locations for the central engine for their orchestration is even harder. Even if such an engine exists, relying on central engines and/or individual big-name vendors would be subject to issues like availability, scalability, reliability, censorship, policy-dependence etc. [12]. Therefore a decentralized approach to open service orchestration would be more attractive to a wide range of next generation mashup applications. In figure 1.1, edges marked as 4 show an example of this kind of orchestration where service providers or other intermediate agents orchestrate the composition as collaborative tasks.

In this thesis work, we use continuation-passing messaging for decentralized orchestrate of open web services. With this approach, orchestration activities are carried out at places close to the open services and no resource is allocated in the distributed environment prior to the execution of the composite application.

1.4 Problem statement

In an open distributed environment, a lot of things can happen beyond the control of anybody. Open services can come and go. Computer systems can crash. Network connections can be torn down. The utmost question this thesis work attempts to answer is:

Q. *Is it possible to perform reliable orchestration of composite open services?*

As open services are outside the administration boundary of any enterprise, the orchestration of composite open services should not rely on central engines. Therefore, we take a decentralized approach to service orchestration.

Decentralized orchestration approaches have already been devised as a research effort to overcome the drawbacks of centralized approach even within enterprise boundaries. Decentralized orchestration, however, is generally regarded as more challenging for certain orchestration management tasks due to absence of global run-time states. In particular, the orchestration of services need to monitor and manage dynamic run-time status and controls of composite services. As this information is needed to utilize further in error or fail recovery, one of the major challenges remains in maintaining this information carefully and effectively.

Existing decentralized approaches rely on the pre-allocation of control and resources prior to the execution of the composite services. The pre-allocated resources take care of the monitoring and management of the dynamic run-time state. In an open environment, however, it could be unfeasible to pre-allocate resources . So our first specific research issues is:

Q1. *Is it possible to orchestrate open services without a central engine and without pre-allocation of control and resources?*

When a software program is running, exceptional conditions may happen. It is more so during the orchestration of composite services where things happen in different places in the distributed environment. For a software program, exception handlers are constructed for certain expected exceptions. When an exception occurs, it is typically propagated to and handled by the corresponding exception handler. With centralized service orchestration, the central engine observes and then handles the exception. With decentralized orchestration that pre-allocates resources and control, an exception is propagated to some pre-allocated controlling entity which then handles the exception. With a decentralized orchestration without pre-allocated of control and resources, our next research issues is:

Q2. *Is it possible to handle exceptions at run time when the execution is dynamically spread around in the distributed environment?*

Robustness of a computer system can be defined as the ability of the system to react appropriately to some abnormal conditions. It is generally known that we can not guarantee [40] to completely prevent failures either by the integrity of the program or by the host environment where the program executes. This is particularly true in an open environment. If we can not prevent failures, then the right mechanism should be able to tolerate them. So our next research issue is:

Q3. *Is it possible to tolerate unexpected failures when the execution is dynamic and distributed?*

1.5 Summary of contribution

In this thesis, we present a decentralized approach to services orchestration called continuation-passing messaging (CPM). Dynamic execution status and control are carried in messages as continuations. The messages also contain exception handlers and recovery plans, called compensation continuations, that are dynamically generated during execution. Our approach tolerate network and system failures with a dynamic replication scheme.

The major contribution of this thesis are presented in the papers [35, 36, 37, 38, 85] outlined in the Appendix.

This thesis has general contribution in the field of distributed computing, as composite web services is a special form of distributed computing. Here we give an overall summary of these contributions:

- We have designed and implemented *continuation-passing messaging* (CPM), a decentralized reliable open services orchestration approach that does not pre-allocate resource and control prior to execution and can handle exceptions at run time [37, 38, 85].
- We have devised a flow-aware dynamic replication approach that tolerates system and network failures [35, 36].
- We have carried out experimental studies and evaluated the performance of our approach [35, 36, 37, 38, 85].

1.6 Brief overview of approach

In our decentralized approach, a network of *orchestration agents* (OAs) collectively orchestrate the executions of processes using *continuation-passing messaging* (CPM) [37, 38, 85]. Service orchestration messages contain information about the flow of control in *continuations* and data in *environments*. The recovery plan

for exception handling is dynamically generated in *compensation continuations*. The initial continuation and environment of a CPM message are generated when an OA starts to orchestrate a composition of services. The message is later on sent to subsequent OAs that independently interpret the messages and invoke the service operations of the appropriate service providers (SPs). New continuations and environments are generated based on the messages being interpreted as well as the outcomes of the service executions.

With CPM, information about the orchestration is usually already spread among multiple OAs. This information, if carefully maintained, could be used to handle occasional unavailability of OAs. This is the key idea behind replicated CPM [35, 36]. One of our primary goals is that the selected set of replicas can reuse as much as possible stored run-time states using CPM in order to keep the run-time overhead of replication as low as possible.

We have developed a prototype to run in a simulator and evaluated our work with simulation.

1.7 Limitations

We decided to focus ourselves on selected important research issues and limit the scope of this thesis work.

We have used simulation to investigate our concepts rather than building a full-featured application prototype. Our primary concern is whether the new approach works at all and how it compares to other relevant related work in terms of performance and scalability. Since our approach is completely new, much of the foundation work must be in place before any realistic application prototype can be built. Attempting to develop a prototype would force us to spend a lot of time on less relevant issues. Simulation also allows us to experiment with different orchestration approaches.

We have concentrated mostly on the control flow part of orchestration of open services. Effective data flow management could be the immediate follow-up of our current work.

We have not worked on security issues, which are clearly highly relevant to distributed application in general and decentralized service orchestration in particular. Other researchers in our research group are working on security issues that could apply to our work [4].

1.8 Dissertation outline

The remainder of the dissertation is structured as follows:

- Chapter 2 presents some background relevant to this this work, including web service composition and orchestration, open services, and challenges.
- Chapter 3 presents a brief overview of *Orchestration Agents* (OA) and an introduction towards the CPM approach to decentralized service orchestration.
- Chapter 4 describes CPM in more detail, including the construction of CPM messages, the interpretation of messages, fault handling etc. We walk through an example to help the reader understand how CPM works.
- Chapter 5 shows a dynamic replication mechanism that tolerates system and network failures during orchestration of open services.
- Chapter 6 evaluates our work with experimental performance studies.
- Chapter 7 concludes and outlines possible future work.

Chapter 2

Background

In a service oriented architecture (SOA), individually shaped services are composed to be collectively and repeatedly utilized to meet specific business goals. In the literature [59], "*orchestration* refers to an executable business process (i.e. a composition) that can interact with both internal and external (web) services. The interactions occur at the message level. They include business logic and task execution order, and they can span applications and organizations to define a long-lived, transactional, multi-step process model."

Services in SOA is normally constrained within the same enterprise boundary. Traditional workflow or business process technologies have been successfully applied to this architecture for the interaction among component services. Orchestration of composite services in SOA is usually carried out by dedicated central engines. However in open service, these interactions usually exceed enterprise or organization boundaries. It is therefore hard to find feasible locations for central engines. The primary goal of this dissertation is to achieve fully decentralized orchestration of open services.

In this chapter, we present the background of our research, namely service composition, open services and decentralized service orchestration. We also discuss research issues, with an emphasis on issues concerning reliability in decentralized service orchestration.

2.1 Composition of services

In general, composition is the process of building a larger structure by combing or assembling smaller components. In our context, these smaller components are services. They are fundamental elements or building blocks for developing large-scale applications. In computing, services can be defined as platform and network independent operations that clients or other services invoke [51].

The technology of business process [80] and workflow [43] is widely adopted for services composition and orchestration. For example, in WS-BPEL [57], the *de facto* standard for web-services composition, where individual web services are composed into BPEL processes. According to [80], a *business process* consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations. Business processes may consist of parts that are carried out by computers and parts that are not supported through computers. A *workflow* is the part of a business process that is carried out by computers [43]. If we limit the activities to services or web services, workflows become service or web service compositions.

In the literature [63], as well as in this dissertation, terms for workflow, business process (or simply processes) and service composition are often used interchangeably. For example, a workflow or process corresponds to a service composition, a task corresponds to a service, workflow enactment corresponds to services orchestration, and so on.

Service composition is most beneficial when currently available services do not fulfill the required functionalities while as a coordinated composition it can perform that. Thus a composition of multiple services can make more capable and powerful applications. Service composition consists of several steps: selection of appropriate services, specification of composition in some execution languages (for example, WS-BPEL, WS-CDL, etc.), verification of the service composition according to the objective and composition requirements, and finally monitor or adaptation of the composition if it is required [60].

Figure 2.1 shows an example composition p. Normally a service (simple or composite service) is specified by an *identifier* (e.g., URL), a set of *operations* and a set of *attributes*. Here in our examples, we have ignored attributes for the sake of simplicity; instead of using URL as an identifier we have used service providers *name* to locate a service.

The example composition consists of invocations to operations a at S_a, b at S_b, c at S_c and d at S_d. p 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. The element (or activity in BPEL's terminology) $invoke(S_a, a)$ means: "run service operation a at S_a.

2.2 Open web services

In general, Web Services can be considered as a way of communication between computer programs using traditional Web technologies namely the HTTP net-

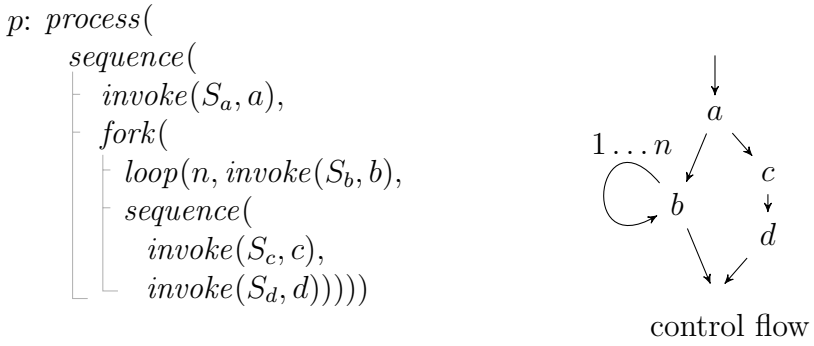


Figure 2.1: An example composition

work (application layer) protocol. The development and standardization of Web Services technology is coordinated by the World Wide Web Consortium (W3C) in the framework of Web Services Activity [78]. According to the Web Services Glossary by W3C [79], the definition of the Web Services is as follows:

> A Web service is a software system designed to support inter-operable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

In the Web Services Activity Statement, it is mentioned that one of the important aspects of Web Services is the ability to combine services "in a loosely coupled way in order to achieve complex operations"[78], meaning that Web services are meant to be inter-operable and extensible. We use the term *open* to emphasize the interoperability, extensibility and autonomy of services in general, and web services in particular.

In recent years, an ever-growing large number of web applications provide open services through published APIs [20].

A new generation of applications are built by combining and integrating the functionalities and data from these open web services. A particular group of such open-service based applications – web applications – are widely known as web mashups. The initial idea of mashup was to mix and mash up search results and to visualize those results in more interesting ways. However, programmers can take a lot of leverage from these open services and can even develop new innovative services and offer them to the website visitors. For example, a composition may handle a whole purchasing service that includes services which calculate the

final price for the products, select a shipper, and schedule the shipment for the order. Again this service composition may itself become services, thus it makes a composition as a recursive operation.

Successful adoption of open-service based applications requires both development-time and run-time support [83]. Development-time support includes the tools for correct service invocation through open APIs and for extraction and conversion of data obtained from the external services. Run-time support, or orchestration of open services, is the conduct of the execution of applications that use these open services. Thus a service orchestration may combine services following a certain composition pattern to provide new service functions.

2.3 Orchestration of open services

Business processes or workflows are typically enacted in a centralized manner. A central engine monitors the progress of the processes, maintains their run-time status and conduct the executions of the tasks. This has been working very well, given the application scenarios of business processes. However, when applied to composition of open web services for a wider range of applications, the traditional business process technology has a number of serious limitations.

The application scenarios of composing widely open web services are significantly different from those of traditional business processes. Traditionally, business processes are primarily constrained within enterprises and many tasks are carried out by human workers. Moreover, business processes usually involve regular routines and have stable structures. The same process is therefore repeatedly instantiated and executed. Web services may not be constrained within enterprise boundaries and the services are typically run purely by software without human intervention. The compositions normally do not involve regular routines and the structures can be dynamic and flexible. Moreover, new Internet-scale applications are appearing in a rapid pace, exemplified by social networks and peer-to-peer data sharing. It is not hard to envision that services composition will eventually be applied in or even across such Internet-scale applications.

Consider the following example where conference organizers would like to build a temporary social network for a conference. If upon registration, a participant provides her interests, a registration report may offer useful services like recommendations for accommodation and sightseeing, people with similar interests (such as research, country, institute, etc.), as well as information about the conference itself: number of submissions, acceptance rate now and earlier, most referenced papers, awards, and so on. Many of these can be constructed by composing existing external services. These can further be the basis of new services, such as collaborations among particular groups of people.

The number of compositions as the above example is potentially unlimited. The compositions can also be very dynamic. Some of those could be built by the conference organizers and be instantiated multiple times; some others could be composed by the participants and might be run only once or even often be aborted half way.

In a centralized approach, the central engine sends messages to service sites for service invocations. When a service is done, or when some fault occurs, a service site sends a message back to the engine, either as a return message, or as a call-back. Information like activity execution order and run-time state is maintained at the central engine. To be applied to Internet-scale applications, this approach is constrained with scalability limits. Furthermore, for such applications, there is hardly a suitable place where a central engine can reside.

Based on these observations, we argue that a decentralized approach could be more feasible for the orchestration of composite open services.

2.4 Dencentralized services orchestration

Over the years researchers have noticed issues and limitations of centralized orchestration approaches and have proposed decentralized approaches. The general idea is that, the orchestration is carried out collectively by a number of engines or agents, spread around in the distributed environment.

We classify decentralized approaches into two groups: instantiation-based and messaging-based. With instantiation-based approaches (for example [8, 22, 55, 56, 68]), a composite service is instantiated before execution. During an instantiation, the resources and control are allocated in the distributed environment based on an analysis of the composition structure. As a common problem to these approaches, resources are allocated even for the parts that are actually not executed, such as some of the alternative paths or when a process rolls back at an early stage.

With messaging-based approaches (for example, [11, 52, 67, 82]), the information for controlling the order of execution is carried along with messages at run time. In the current messaging-based approaches, part of the static specification of the process, for instance represented as mobile code, is carried in messages for service executions.

When instantiation or messaging relies on static process structures for decentralization, they are subject to difficulties for tasks that cannot be properly planned in advance, such as fault handling and recovery. To address these issues, these approaches typically delegate such tasks to a single site [8, 14, 22, 52, 82]. They are thus subjected to the same issues of the centralized ones.

A web mashup, as of today, can invoke open web services and compose them internally using any host programming language. The current web mashups are

not subject to the challenges due to central engines. However, it is the programmer of the individual web mashups to deal with all the lower-level details of composition and management tasks. A means of higher level composition is a natural next major step toward wider adoption of web-services compositions.

2.5 Challenges with decentralized services orchestration

Computer systems are subjected to performance and reliability challenges. Decentralized services orchestration is no exception. Scalability is an important performance measurement. In general, we regard a system as scalable if it can handle the addition of requests and resources without significant additional cost and complexity or loss of performance. In decentralized orchestration, absence of a centralized engine reduces the possibility of a potential performance bottleneck. However, inappropriate design of a decentralized system can also lead towards potential deadlocks or non-optimal usage of system resources [15]. In Chapter 6, we compare the scalability of three orchestration approaches: centralized with central engines, decentralized with continuation-passing messaging, and decentralized with instantiation of control prior to the execution of a composition.

In computer science, *dependability* and *reliability* are often used interchangeably as both are related with the fault tolerant behavior of the system, though conceptually they have subtle differences. In fact, dependability covers useful requirements for distributed systems like: availability, safety, maintainability, and reliability. In our context, dependability is a quality of the delivered service so that other services can trust or rely on the service and may build other services based upon this service. Reliability refers to the property of a system or component that can perform its functionalities continuously without any failure [73].

Therefore, reliability of a service becomes always an issue while services or systems depends on other services or systems, as is the case of composition and orchestration of services.

The main impairments to establish a reliable service are: faults, errors and failures. A system *failure* occurs when its delivered service differs from the expected service. If the system is an application that uses another service, a failure may occur due to some erroneous condition met in the application host, the service providers site or in the network infrastructures. In this context, an *error* is that part of the service site or network state which is liable to lead to the failure. The cause of an error is a *fault*. In other words, the failure of a component is a fault that causes an error and leads to a failure of an entire system. In the context of service orchestration, faults can occur in any of these places: at a service, at an orchestration element or at the communication network.

There are basically two approaches toward system reliability: fault prevention and fault tolerance. Fault prevention aims at reducing the possible number of faults. Fault tolerance aims at recovering from errors. We work towards fault-tolerant service orchestration, since faults will occur in a distributed environment beyond our control.

2.5.1 Fault at services

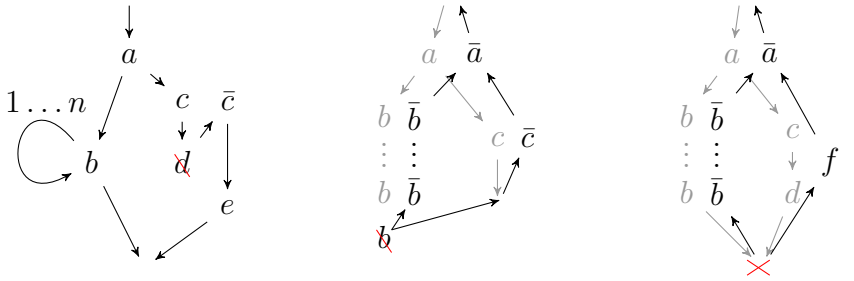
An observable service failure may occur in one of two ways: either the service site throws a fault (also known as an exception), or it does not respond to a request at all. An observable service failure can be handled through the fault-handling mechanism of the service composition mechanism.

In BPEL, services can be composed into a hierarchical structure, as nested scopes, and fault handling is associated with scopes. Let us consider a hierarchical service composition structure. Individual services are composed into scopes, lower-level scopes are composed into higher-level scopes. A top-level scope is a composite service, which can either be adopted as an application or as an open service.

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

Figure 2.2: An example composition with fault handling

Figure 2.2 shows an extension of the composition in Figure 2.1. The new composition now includes the concept of scope and fault handling. We assume that operations a, b, c and d have reverse operations $\bar{a}$, $\bar{b}$, $\bar{c}$ and $\bar{d}$. In BPEL [57], *compensation* means logical *rollback* and reverse operations are called *compensation operations*. The element *invoke*$(S_a, a, \bar{a})$ in Figure 2.2 means: "run service operation a at service site S_a; if the composition p has to be rolled back due to an



(a) rolling forward (b) rolling back (c) rolling back

Figure 2.3: Control flow of example composition

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 a composition construct that is not understood by S_a. S_a only understands either $invoke(a)$ or $invoke(\bar{a})$.

The top-level scope has a fault handler of *any* faults. Upon a fault of *any* type, the scope simply rolls back the service operations that have successfully executed within the scope so far.

The nested scope has a fault handler, also of *any* faults. It first rolls back the executed service operations and then invokes service operation e to *roll forward* the current scope.

The nested scope has also a *compensation handler* that invokes service operation f. The compensation handler provides the rollback plan for the situations where the scope has successfully completed but the top-level scope fails and it has to be rolled back anyway.

Figure 2.3 shows the control flows after a fault. If a fault occurs within the nested scope, according the the fault handler of the scope, the scope first rolls back the finished service c and then rolls forward by running service operation e (Figure 2.3-a). If a fault occurs in the top-level scope, the currently completed execution is rolled back. There are two different cases. If the nested scope has not completed when the fault occurs, the reverse operation $\bar{c}$ of the completed operation c is executed (Figure 2.3-b). If the nested scope has completed, the operations in its compensation handler is executed. That is, instead of running the revrse operations $\bar{d}$ and $\bar{c}$, a new operation f is executed (Figure 2.3-c).

In decentralized services orchestration, scopes, fault handlers and compensation handlers work together to handle faults during the execution of composite services. If a scope fails, the predefined compensation handlers are supposed to be activated and undo the completed activities. Furthermore, exception handlers perform the task of forward progression which in turn leads towards the termination of the process [33]. Error handling mechanisms in decentralized services orchestration has several challenges:

Managing control context

Run-time monitoring of compositions has always been difficult and is acknowledged as a significant and challenging problem [6]. Managing the run-time information and control context of the composite service (as well as of the component services) while the execution is flowing from one place to another is a challenging task to perform.

Propagation of faults

Fault propagation is important for compositions as a fault in one service can lead to a failure in the whole composition. Therefore, we need to employ mechanisms to propagate and notify about the fault to appropriate orchestration elements.

Generating recovery plans

As compositions include a series of service invocations, a failure in one service may need to undo the previously completed services. As services are dependent on each other in sophisticated ways (intra- and inter-scopes), generating and automatizing recovery plans for the service composition can be complicated.

2.5.2 Fault at orchestration elements

Fault handling of composite services works only when the service orchestration infrastructure is still working. However, faults may arise in the orchestration infrastructure itself. For example, an orchestration element or component may crash. To tolerate faults of the orchestration infrastructure itself, we have to introduce redundancies [28, 61].

One common way of tolerating the failure of a component is replication. That is, the system uses multiple instances of the same component (replicas) and these instances fail independently. The run-time state of the component is replicated among these instances. When some of the instances fail, the rest instances that are still working can continue to serve the function of the component.

There are two general replication approaches. In a *primary-backup* approach [2, 7, 10], a primary instance of a component is backed up by a number of replicas. During normal operation, the primary instance interacts with the backups to maintain some level of data and state consistency. Whenever the primary instance fails, one of the backup instances takes over the responsibility and continues the function of the component. [2] presents an early single-primary multiple-backup strategy. When the primary instance receives an incoming request, it propagates the request to each of its backups following the same order as it has received those messages. The primary instance does not reply the request until it has propagated

that request to at least one of its backups. In case the primary instance fails, a backup is elected as the new primary. The new primary takes leverage from the necessary information it has received from the previous primary and continues the rest of the operation. The system uses request sequence numbers to assure that non-idempotent operations are performed exactly once. If a backup instance fails, it is removed from the backup list.

Unlike the primary-backup approach, in an *active replication* or *state-machine* approach, all non-faulty instances of the same component actively serve the same incoming requests [17, 66]. To ensure correctness, all non-faulty replicas receive and process the same sequence of requests in the same relative order. To tolerate fail-stop faults, any replica's output can be chosen. To tolerate Byzantine faults, a majority consensus of the replicas' output is necessary.

Replication approaches as discussed above are mostly applicable to client-server systems. Rollback recovery protocols introduce another form of redundancy to long-running applications where multiple processes collaborate through message passing.

In *rollback recovery* protocols, each process has access to a *stable storage* that survives all tolerated failures. During normal operations, processes periodically record their run-time and communication states to their stable storage. When a process fails, the system restarts the process and resume the operation from a recorded intermediate state, thereby reducing the amount of lost work and computation. This saved recovery information, also known as *checkpoints*, usually includes participating process' states. Checkpoint-based protocols and log-based protocols are the main variants of distributed rollback-recovery [19].

Checkpoint-based protocols [16, 61] require the processes to periodically record checkpoints. The frequency and nature of the recording depend on the pattern of the coordination among the processes. One of the requirements of this approach is that in faulty situations, all processes need to rollback to their most recent global consistent state, even for the non-failed processes. Log-based recovery approaches [3, 72] record additional run-time information that allow a failed process to replay the same operations from a checkpoint, so as to avoid the surviving processes to rollback.

With decentralized service orchestration, several orchestration elements, called orchestration agents in our system, jointly conduct the execution of a composition of services. These orchestration agents are distributed in the network. We assume that the network of these orchestration agents is established on a voluntary basis. These agents can be up and down regularly. We cannot assume that a failed agent be able to restart within a given time frame.

Some aspects of such a system impose both special challenges and opportunities with respect to reliability.

Moving targets

As we aim at an approach that does not pre-allocate resources prior to the execution of a composition, the responsibilities of the orchestration tasks are assigned to dynamically selected orchestration agents. These agents jointly orchestrate the composition by exchanging and interpreting orchestration messages. The run-time state of the orchestration moves and updates from agent to agent. Each agent keeps only some partial state that is just enough for certain particular part and stage of the orchestration.

To keep track of the dynamic and distributed state of the orchestration is like shooting at a moving target. There are challenges to reliably monitor and manage the progress of the orchestration. Challenging issues include: Who is responsible for a particular orchestration task? Who is responsible for monitoring the status of an agent? If an agent is detected to be unavailable, which other agents should be notified and who should take over the remaining task assigned to that agent?

In order to be able to take over the tasks assigned to a failed agent, the system must be able to restore the state information maintained by that agent. That is, the necessary information must be replicated somewhere. So the further challenging issues include: What kind of replication mechanism is appropriate for our system? Where should certain state information be replicated? How to maintain the replicated information?

Distributed states

With decentralized orchestration of services, the orchestration state is distributed among a number of orchestration agents. This imposes challenges, as discussed above. In fact, it also presents interesting properties that we could explore.

When an agent has finished its part of the orchestration task, it propagates the responsibility to the following agent. Now these two agent have overlapping state information about the orchestration. So some replication is already in place, for free! In addition, different orchestration agents may have already been collaborating due to the composition structure or for the purpose of fault handling. For example, agents may relate to each other due to hierarchical dependencies or for handling of parallel branches.

Therefore we might be able to benefit from these properties to enhance the reliability of service orchestration. The challenge is: How?

2.6 Summary

In this chapter, we first presented some background about open services, composition of open services and orchestration of open services. We argued that decentralized orchestration is more feasible for open services. We then categorized decentralized orchestration approaches into two main groups: instantiation-based and messaging-based, and proposed that message-based approaches is more suitable for orchestration of open services.

Then, we discussed some of the major challenges with decentralized orchestration, with emphasis on reliability and fault handling. In the context of service orchestration, faults can occur in any of these places: at a service, at an orchestration element or in the communication network. We went into some depth on reliability issues with decentralized orchestration, including fault detection, fault propagation and fault handling, as well as fault tolerance mechanisms for distributed systems including rollback recovery protocols and replication.

Chapter 3

Approach Overview

In the context of services composition, decentralized orchestration system consists of multiple orchestration elements that collaborate with each other without the necessity of a central coordination entity. Here each of the orchestration element plays nearly an equal role in orchestrating the execution of the compositions. The goal of establishing the decentralized orchestration system is to fulfill all the requirements of the centralized orchestration system by utilizing the capabilities of a set of orchestration elements.

The orchestration elements in our system are called *orchestration agents*. Generally a software agent is an entity which is capable of performing flexible and autonomous actions in order to accomplish their design goals [65]. Flexible autonomous agents have already been used in various application domains ranging from autonomous control of spacecrafts to personal digital assistance. In our thesis, a network of orchestration agents collaborate to orchestrate the execution of open services.

This chapter presents a high-level overview of our approach to decentralized orchestration of open services. More details are presented in the subsequent chapters.

3.1 System model

Open services are provided by *service providers* (SPs) in terms of operations in their public APIs. A *service-based application* (SA), also known as a *service composition* in the literature, consists of invocations to a number of service operations in a prescribed manner. *Services orchestration* is the conduct of an execution of an SA.

In our orchestration approach, a network of *orchestration agents* (OAs) jointly orchestrate the executions of SAs using a particular mechanism called *continuation-*

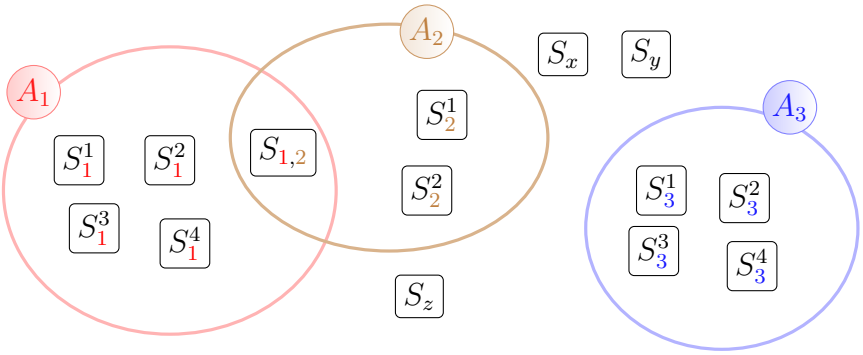


Figure 3.1: SPs, OAs and OA coverages

passing messaging (CPM). The OAs invoke the service operations on behalf of the SAs and are responsible of moving forward the execution of the SAs to the other OAs. In addition, they handle exceptional conditions upon the occurrences of errors.

An OA *covers* a number of SPs. A general criterion for an SP to be covered by an OA is that the geographic distance, and hence the delays of messages, between the OA and the SP is short. To invoke a service, it is advantageous performance-wise to choose an OA covering the corresponding SP.

Figure 3.1 illustrates SPs, OAs and coverage of OAs. As shown in the figure, at a specific moment, SPs may or may not be covered by OAs and OAs may have overlapping coverage. For example, at this particular moment S_x, S_y and S_z are not covered by any OAs; meanwhile $S_{1,2}$ is covered by both A_1 and A_2.

SPs become covered by OAs either by registration to specific OAs or through a learning process (see Section 3.5).

An OA can run on a dedicated server, such as provided by a cloud provider. Alternatively, an SP may volunteer to provide an OA as well. Providing an OA may make an SP's service more attractive. For example, if either an SP or the cloud hosting the SP has an OA, repetitive invocations to SP's services may appear to be much faster, as shown in our experiment in Chapter 6.

An SP may be unavailable, due to disconnection or system crashes, and does not respond to invocations. An SP may also return an error. We assume that business critical services 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 (Chapter 5) enhances the availability of the orchestration when the OAs are subject to such unavailability.

3.2 Continuation-passing messaging

With CPM orchestration, information like operation execution order and SA-aware data is carried in orchestration messages, called *CPM messages*, in terms of continuations and environments. A *continuation* is a stack of activity elements, such as *scope*, *fork*, *invoke* that will be carried out, beginning from the head of the stack. An *environment* contains information of activity status and SA-aware data.

The OAs interpret the received CPM messages and conduct the execution of services. New continuations are generated based on the messages being interpreted as well as the outcomes of service executions. The outcomes of the service executions and the remaining activities of the process are carried in new CPM messages to the subsequent OAs.

Therefore services orchestration is actually a sequence of message exchanges and interpretations by the involving OAs.

An SA specifies how exceptions are handled with fault handlers associated with scopes. To facilitate exception handling during the execution of SAs, CPM messages also contain *compensation continuations*, which are rollback plans automatically generated during the execution of SAs.

Figure 3.2 shows the overall structure of an OA. The message handler dispatches the incoming messages to the corresponding components. There are three types of messages: CPM messages for the orchestration of the execution of SAs, scope messages for the management of SA scopes, and OA routing messages for the management of OA networks.

When an OA is asked to conduct the execution of an SA, it generates a CPM message with initial continuations and environment. The activity elements in the continuations are assigned with OAs according to the information offered by the *OA routing component*. Later, the knowledge from the other OAs may help choose better alternatives for the assignment of OAs. The CPM message is then interpreted.

The message interpreter interprets an incoming or a local CPM message according to the head element of the continuation. The following may happen during the interpretation:

- In some cases, a message can be interpreted alone. In other cases, multiple messages must be available to be further interpreted, for example, when

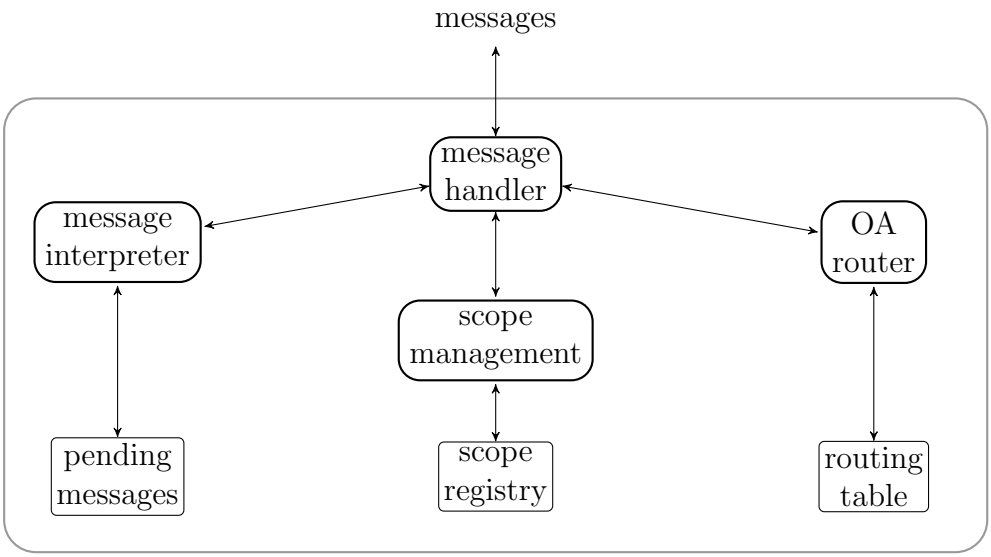


Figure 3.2: Structure of an Orchestration Agent

messages from multiple parallel branches join. In the latter cases, the first arrived messages are put in the *pool of pending messages*. 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 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 corresponding SP and waits for the result. The message is further interpreted according to the out-come of the invocation.

An OA may also be a *scope manager* and maintains some status information about scopes in its *scope registry*. The main task of scope management is fault handling. When an OA throws a fault, it notifies the scope manager about this, which in turn propagates this notification to the other branch OAs of the scope. In order to achieve this, every CPM message contains the information about the scope and scope manager, and every scope manager maintains in its scope registry the current OAs of all branches for each scope. When an OA sends a message to another OA so that the current branch is passed forward to the next OA, it also sends a message to the scope manager, which keeps its scope registry up to date.

OAs also exchange messages for the management of OA networks. The OA router handles the OA management messages (see Sections 3.4 and 3.5).

3.3 CPM by example

Let us use the example composition p in Figure 2.2 to see how CPM works.

To start the execution of p, the SA provider S_p requests OA A_p for the execution of p by sending the message $orch(p, S_p)$.

When receiving the message, S_p converts p into an initial CPM message, where it assigns OAs to the corresponding activities according to the information in the OA routing table. Assume that SP S_a is not covered by any OA and SPs S_b, S_c and S_d are covered by OAs A_b, A_c and A_d, the initial CPM message looks like

$$orch^{A_p}(scope^{A_p}(invoke^{A_p}(S_a, a, \bar{a}) \cdot fork(loop(n, invoke^{A_b}(S_b, b, \bar{b})), \dots)))$$

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* activity.

A_p then start orchestrating the execution of p by interpreting the CPM message.

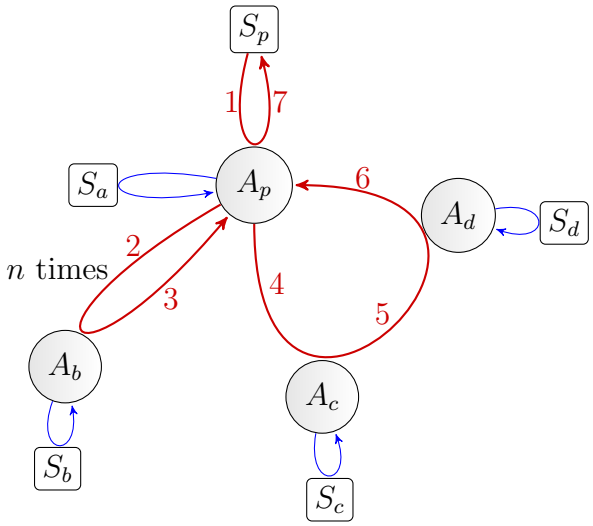


Figure 3.3: Service invocation and orchestration messages

Figure 3.3 shows the messages among OAs for the orchestration of the example SA p. There are three types of messages for services orchestration: CPM messages (red lines), service invocation messages (blue 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 availability status of other OAs.

In Figure 3.3, message 1 is $orch(p, S_p)$, the message from S_p that requests for orchestration. The initial CPM message $orch^{A_p}(scope^{A_p}(-))$ is a local message at A_p. Note that local messages are not shown in the figure.

In some cases, a message can be interpreted alone. For example, the CPM message $orch^{A_p}(scope^{A_p}(-))$ is interpreted into $scope^{A_p}(-) \cdot eorch^{A_p}(-)$, which in turn is interpreted into $invoke^{A_p}(S_a, a, \bar{a}) \cdot fork(-) \cdot eos^{A_p}(-) \cdot eorch^{A_p}(-)$. 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 3 and 6 from the two parallel branches join. In this case, A_p puts the first arrived message, say message 6, in the pool of pending messages. When message 3 from the other branch arrives, A_p interprets messages 3 and 6 together and the execution of p moves on.

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 activity 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 message pool. For example, interpreting message

$$invoke^{A_a}(S_a, a, \bar{a}) \cdot fork(-) \cdot \ldots \cdot eos^{A_p}(-) \cdot orch^{A_p}(-),$$

A_a sends $invoke(a)$ to S_a and puts message

$$wait^{A_a}(S_a, a, \bar{a}) \cdot fork(-) \cdot \ldots \cdot orch^{A_p}(-)$$

in its message pool. A_a will later interpret the *wait* message according to the outcome of the the service a.

An OA may also be a scope manager and maintains some status information of each branch in its *scope registry*. In particular, a scope manager keeps track of the current location of each enclosing branch. In our example, when the second branch moves from A_c to A_d with message 5, A_c informs the scope manager A_p of the move with a scope management message.

The following table lists the continuations in the remote CPM messages as shown in Figure 3.3. 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.

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	$invoke^{A_c}(S_c, c, \bar{c}) \cdot invoke^{A_d}(S_d, d, \bar{d}) \cdot eos^{A_p}(id_2, -) \cdot \kappa$
5	$invoke^{A_d}(S_d, d, \bar{d}) \cdot eos^{A_p}(id_2, -) \cdot \kappa$
6	$eos^{A_p}(id_2, -) \cdot \kappa$
7	$eorch(S_p)$
κ	$join^{A_p}(id_j, 2) \cdot eos^{A_p}(id_1, -) \cdot eorch^{A_p}(S_p)$

A *join* activity 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* activity marks the end of a scope and encapsulates necessary information for exception handling. The general form of an *eos* activity is

$$eos^{A}(id, \kappa, h_1, h_2 \dots),$$

where A 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. It is automatically generated during the orchestration. The table below lists the compensation continuations in the *eos* activities of the remote CPM messages. Notice that in messages 4, 5 and 6 there are two compensation continuations, one in the *eos* element of the nested scope, the other in the *eos* element of the top-level scope.

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

If the execution of service b fails, A_b will catch an exception. It then runs the corresponding exception handler in the enclosing *eos* activity and at the same time notifies the scope manager A_p of the exception. A_p then propagates the exception to the other branch.

Chapter 4 discusses in detail how CPM messages are interpreted.

3.4 Organization of an OA network

The organization of an OA network is based on the *distances* between OAs. Distances can be measured in terms of communication delays or number of IP hops. Regardless of the measure, long distances imply long delays. An OA has a *ping* operation that can be called to measure the distance with the caller.

OAs organize themselves into *OA flocks* according to their distances. A small subset of OAs near the geo-graphical center of a flock are the *head OAs* of the flock. The *radius* of a flock is the longest distance that is allowed from any OA of the flock to a head OA. The *distances between flocks* are measured with the distances between head OAs of the flocks.

The OA management component of an OA (Figure 3.2) maintains the following tables:

- T_{OA}: OA $\rightarrow$ flock
- T_{SP}: SP $\rightarrow$ OA(s) or flock
- T_{flock}: flock $\rightarrow$ head OAs, radius, distances to flocks
- $T_{distance}$: OA $\rightarrow$ distance to OAs of the same flock.

T_{OA} tells which flock an OA belongs to. T_{SP} provides information about which OA(s) or flock an SP is covered by. The distance information is maintained at two levels: T_{flock} provides distances between flocks (as well as the flocks' head OAs and radii); $T_{distance}$ provides distances between OAs of the same flock.

Maintaining distances at two levels makes the network more scalable for OA routing. The distance granularity at flock level is usually sufficient to the assignment of SPs to OAs for the orchestration of SAs. Therefore an SP can be covered by a flock, so that any OA of the flock can be assigned to an invocation on an operation of the SP. When fine grain optimization of performance is needed, an OA closer to the SP can be chosen to invoke the operations of the SP.

Routing information is propagated at two levels. At the inter-flock level, updates of the T_{flock}, T_{OA} and T_{SP} are propagated among head OAs of different flocks. At the intra-flock level, updates of all tables are propagated among the OAs of the same flock.

When a new OA joins the OA network, it obtains a T_{flock} from any OA and measures the distances with all flocks. If the new OA is within the reach of the nearest flock (by comparing the distance to a head OA and the flock radius), it joins that flock. Otherwise, a new flock is created. A flock is split if it is overpopulated.

3.5 Covering SPs

An SP can register to the OA network to be covered by proper OAs. There are two options for the registration: an SP can request to be covered by specific OA(s), or it can enable response to incoming ICMP (Internet control message protocol) *ping* messages for the OA network to learn about it.

With the former option, the SP is covered immediately by the specified OA(s). This option is useful when, for example, an application or organization sets up an OA in the same subnet of the SP, or when a cloud provider offers an OA to cover all services running in the same cloud. Providing such OAs makes the services under their coverage more attractive. For instance, when the services in the subnet or cloud are invoked in loops, the total delay of the loops is significantly shortened (as our performance study in Chapter 6 shows).

With the latter option, we present two simple algorithms for the OA network to learn about the SP and find an appropriate flock for the SP.

In the first algorithm, a head OA from each flock calls the provided *ping* operation of the SP to measure the distance. The SP is then covered by the OAs of the nearest flock (if the SP is within the reach of that flock).

In the second algorithm, the following *learnWithPing* operation is called on a series of head OAs of different flocks to learn about the SP.

Figure 3.4 shows the steps to learn about the SP S. The learning starts at any flock (flock headed by OA A_1 in the figure). A_1 first measures the distance to S and gets d_1 (line 3 of the algorithm). S is covered by the flock headed by A_1 if d_1 is less than the radius of the flock (lines 6–8). Otherwise, all flocks that are d_1 away from A_1 (plus/minus their radii, shown with the pink circle in the figure), obtained as the result of method *candidates_ping* on the table T_{flock} (line 10), are candidate flocks for S. Pick any flock (flock headed by OA A_2 in the figure) from the candidates (line 16) and continue with the process (line 17). Flocks that are both d_1 away from A_1 and d_2 away from A_2 (at the intersections of the two circles), are the new candidates. If there are sufficient OAs for all locations, this algorithm normally terminates within 4 tries. In the figure, OA A_3 is found to cover S with 3 tries.

Both the proposed algorithms require that the network is not congested so that the measured distances are valid. The first algorithm is more straightforward.

```
1: procedure learnWithPing(sp, min_dist = ∞, candidate_flocks = all_flocks)
2: {
3:     dist ← sp.ping()
4:     if dist > min_dist then
5:         return nil
6:     else if dist ≤ self.radius then
7:         T_SP.add(sp, self)
8:         return self
9:     else
10:        candidates ← candidate_flocks ∩ T_flock.candidates_ping(self, dist)
11:        if candidates = {} then
12:            return nil
13:        else
14:            result ← nil
15:            while result = nil ∧ candidates ≠ ∅ do
16:                next_flock ← candidates.pop()
17:                result ← next_flock.learnWithPing(sp, dist, candidates)
18:            end while
19:            return result
20:        end if
21:    end if
22: }
23: end procedure
```

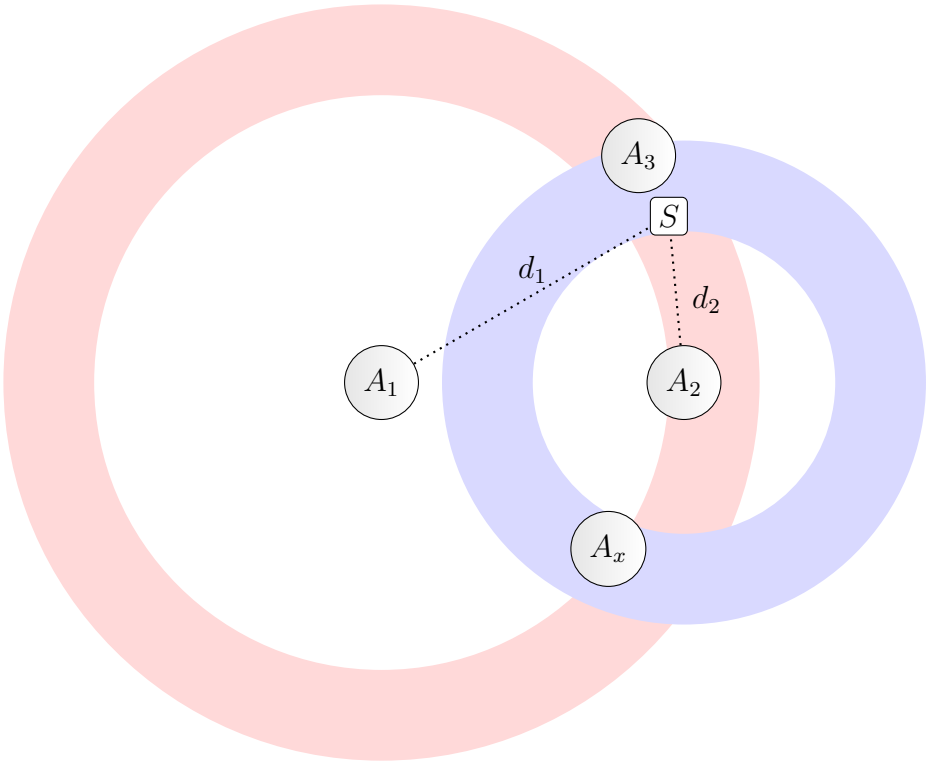


Figure 3.4: Steps of *learnWithPing* to learn about an SP

It guarantees that the nearest flock is chosen to cover the SP. If, however, the number of flocks is large, this may involve a lot of calls to the ping operation of the SP. Furthermore, the distances that are measured must be gathered at a selected OA so that the final decision could be made.

The second method involves only a few head OAs in sequence. It takes much less messages and is therefore much cheaper. It may however not find the nearest flock. So the result is sub-optimal.

OAs can also learn, dynamically during orchestration, about the SPs that are not registered. The learning is embedded in loops of SA executions. In each of the first iterations of the loop, a different head OA is assigned to the invocation to the operations of the SP to be learned. An OA learns the distance to an SP by measuring the response time of the service operations. However, the *learnWithPing* algorithm will not work in this case, because the response time includes both communication time and operation execution time.

We present here a *learnInOrch* algorithm that takes operation execution time into consideration. The *learnInOrch* algorithm assumes that the network is not congested and that operation execution time is stable during the learning period. We think this is not a very strong assumption, because the learning period is short

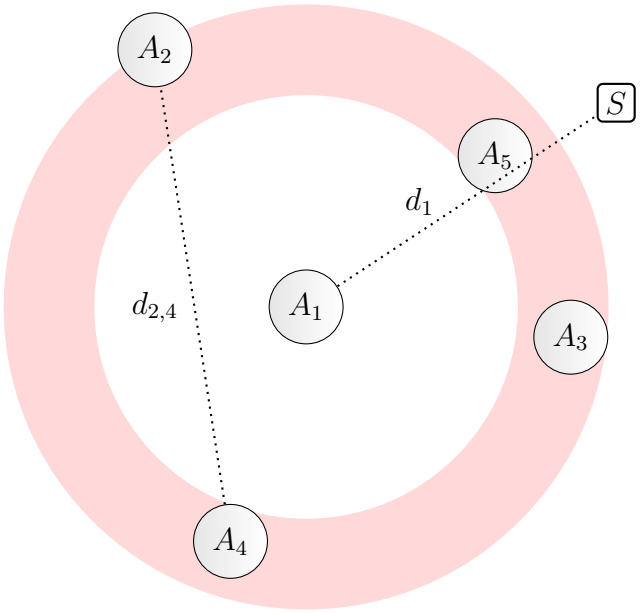


Figure 3.5: Steps of *learnInOrch* to learn about an SP

and is embedded in an SA loop. The different invocations of the same operation within a loop typically have similar contexts and therefore the execution time typically does not vary significantly.

Figure 3.5 illustrates how the *learnInOrch* algorithm works. The learning starts with any flock (flock headed by OA A_1 in the figure) in a first iteration of a loop. A_1 measures the response time of the executed operation. It makes an initial guess of the execution time t_S and the measured distance d_1 excludes t_S. In the next three iterations, the invocations of the same operation are assigned to new flock headers A_2, A_3 and A_4 respectively. These flocks are chosen with the properties $d_{1,2} \approx d_{1,3} \approx d_{1,4} \approx d_1$ and $d_{2,3} \approx d_{2,4} \approx d_{3,4}$ where $d_{i,j}$ is the distance between flock header A_i and A_j and can be obtained from the table T_{flock}. That is, flock headers A_2, A_3 and A_4 are evenly distributed on the circle centered at A_1 with radius d_1. These flock headers then obtain the distances d_2, d_3 and d_4 from S, all excluding t_S. Now choose flock headers A_2 and A_3 that are closer to the S, that is $d_2 < d_4$ and $d_3 < d_4$. Then choose flock header A_5 that is closest to S for the next iteration. A_5 is near the intersection of the line between A_1 and S and the same circle around A_1 with radius d_1. That is,

$$d_{1,5} \approx d_1 \wedge d_{2,5} < d_{2,3} \wedge d_{3,5} < d_{2,3} \wedge \frac{d_{2,5}}{d_{3,5}} \approx \frac{d_2}{d_3}.$$

If the response time of the service invoked from A_5, t_5, is close to t_S, the initial guess of t_S happens to be a good one and the flock headed by A_5 will be covering S. Otherwise, the difference between t_5 and t_S would suggest a new guess and

the learning process continues. We found that A_5 is usually already very close to S and a second guess is sufficient to find the right flock to cover S. Therefore the right flock is found with 5 or 10 tries.

3.6 Related work

In this section, we discuss briefly research work related to OA network. We discuss research work related to CPM in the next chapter after we have presented CPM in detail.

Organization of an OA network is related to at least two technology areas: peer-to-peer (P2P) system and content delivery networks (CDNs).

P2P systems are widely regarded as having the properties like highly scalability, resilience to faults, resistance to surveillance and low cost of ownership [47].

P2P data sharing is one of the most popular application area of the P2P technology. Routing to data items is either unstructured, such as with random walking [30], or structured, such as through an overlay network with a distributed hash table (DHT) [26, 32, 62, 64, 70]. In their most basic forms, data are identified using storage keys and routing is conducted with the hash structure of the keys. To improve performance, proximity-base optimization is often applied using measures like round-trip delay time [18, 32, 46, 87].

The OA network is different from P2P data sharing in that distance (i.e. latency) to SPs is the primary criterion of OA routing.

A CDN deploys a large number of edge servers throughout the Internet. When a user requests some content, a nearby edge server, rather than the content provider, is selected to deliver the content to the user. This may significantly reduce the delivery time to the user. The basic technique for edge server selection is well established and involves simply identifying and resolving content providers' hostnames [76].

The selection of an OA is just the opposite as the selection of an edge server in a CDN: the OA should be as close to the service provider as possible, whereas the edge server should be as close to the user as possible. The techniques of edge server selection in CDN can be adopted in OA selection.

In the context of services orchestration, OSIRIS [68] and OSIRIS-SR [71] share some similar design goals and principles as ours, including achieving high scalability and reliability through decentralized orchestration and replication. In OSIRIS-SR, execution nodes, which correspond to SPs and OAs in our model, are organized in a ring, like the ones for P2P data sharing. The identifiers of the nodes in the ring determine the responsibilities of monitoring of execution and handling of faults. This determinism makes management of the network easier. It does not explore the proximity of nodes and process structures for efficient

process execution, as we do.

3.7 Summary

Our system consists of *service providers* (SPs) and *orchestration agents* (OAs). A *service application* (SA) is a composition of services provided by SPs. A network of OAs jointly orchestrate the executions of SAs, using a particular mechanism called *continuation-passing messaging* (CPM). That is, OAs orchestrate the execution of an SA through exchanging and interpreting CPM messages.

The main elements of CPM messages include continuations for flow control, environments for run-time data, and compensation continuation for fault handling and recovery. We presented a high-level description of CPM through an example.

An OA covering an SP is responsible for the invocation of services provided by the SP. The geographic distances between the SP and the OAs determine the designated OA. In order to maintain a stable OA network, OAs maintain several tables and propagate routing information accordingly. We devised a simple organization of OA networks and some experimental OA selection algorithms. As our focus is mainly on CPM service orchestration, we do not explore further on OA network management .

Chapter 4

CPM in Detail

As continuation-passing messaging (CPM) is one of the fundamental components of our thesis, here we introduce the related concepts and the details about services orchestration in CPM. Conceptually, the core idea behind CPM is based on *explicit message passing* mechanism. That is, orchestration agents collectively orchestrate the execution of composite services by exchanging and interpreting messages. The key, therefore, is message interpretation.

Our service composition language is based on BPEL [57], though application of CPM is not specific to BPEL. In BPEL's terminology, a SA (service application, i.e. a composition) in our model (Section 3.1) is called a *process*.

4.1 Messages

We use the following conventions in the representation of messages:

sites: s	identifiers: id	names: x, y, op, f, l
values: v	expressions: e	activities: $a, ba, ca, pa, eh, fh, ch$
links: ln	environments: env	continuations: c, k

We try to use representative names. For example, ba for BPEL activity, ca for continuation activity, eh for event handler activity, and so on. For continuations, we use c for normal continuations and k for compensation continuations.

Figure 4.1 shows the constructs of messages in a variant of Backus-Naur Form (BNF), where $a^?$ indicates zero or one instance of a, a^* a sequence of any number of instances of a (separated by ','), and a^+ a sequence of one or more instances of a.

A *message* consists of a continuation and an environment, which correspond to the control and data parts of a process respectively. A *continuation* is a stack of activities. An activity is either a *BPEL activity* as specified in the

$$
\begin{array}{rll}
& x, y, op, f, l & \text{(name)} \\
ln & ::= \; l : source \mid l : target & \text{(link)} \\
ba & ::= \; \tau \mid pa \mid assign(x, e) & \text{(BPEL activity)} \\
& \mid scope(x, y, \ldots, ba, \{f : fh\}^*, ch^?, eh^*) & \\
& \mid sequence(ba^+) \mid flow(ba^+, l^*) \mid if(e, ba, ba) \mid while(e, ba) & \\
& \mid receive(op) \mid reply(op) & \\
& \mid invoke(s, op, f : fh^*, ch^?) \mid send(s, op, f : fh^*, ch^?) & \\
& \mid throw(f^?) \mid rethrow(f^?) \mid compensate & \\
& \mid (ba, ln^+, e) & \\
eh,fh,ch & ::= \; ba & \text{(handler)} \\
pa & ::= process(x, y, \ldots, ba) & \text{(process)} \\
ca & ::= eop \mid eot(s, \{f : fh\}^*, eh^*) & \text{(continuation activity)} \\
& \mid eos(s^?, id, \{f : fh\}^*, ch^?, eh^*, l^*, c) \mid eosk(s^?, id) & \\
& \mid join(s, id, e) \mid notify(s, id, f) \mid lnk(l, e) & \\
a & ::= ba \mid ca & \text{(activity)} \\
c & ::= \bot \mid orch^s(pa, s_c) \mid eorch(s_c) \mid inst(ba, s) \mid a^{s^?} \cdot c & \text{(continuation)} \\
m & ::= < c, env > & \text{(message)}
\end{array}
$$

Figure 4.1: Constructs of messages

BPEL standard [57] or a *continuation activity* defined particularly for specific orchestration tasks. An activity a in a continuation can be optionally associated with a site, written as a^s, where s is the site of an OA (orchestration agent) for the activity. *id*s uniquely identify the constructs of specific process instances. They could be either the correlation sets[1] as specified in the BPEL standard, or identifiers automatically generated by message interpreters.

We do not attempt to be exhaustive on the treatment of BPEL constructs. For example, we only deal with the installation and teardown of event handlers and leave the behavior of event handlers untouched.

We will describe the individual activities when we explain the interpretation of the corresponding messages in the subsequent sections. For the sake of con-

[1] During its lifetime, a business process instance typically holds one or more conversations with services. A *correlation set* specifies the correlation of the messages involved in a conversation.

venience, we name a message with its continuation's head activity. For example, if the head activity of a message is a *sequence* activity, we call the message a *sequence* message.

Message interpretation is defined by *interpretation rules*. Below is an example rule, named Rule (oa), standing for the orchestration-agent rule:

$$< a^s \cdot c, env >_{s_c} \xrightarrow[s]{oa} < a \cdot c, env >_s \qquad \text{(oa)}$$

A subscript of a message indicates the site of an OA at which the message is interpreted. When it is clear from the context, we omit the subscript. We use $\xrightarrow[site]{rule}$ to represent a message interpretation. The text above the arrow is the interpretation rule applied. If there is text under the arrow, it indicates the site to which the message is sent.

In Rule (oa), $a^s \cdot c$ is a continuation. a^s is the head of the continuation and c is the tail. The messages at both sides of the interpretation rule are an a-message. The Rule (oa) says that when site s_c interprets an a-message assigned to OA s, it sends the message to s.

4.2 Environment and contexts

BPEL allows different visibility of process-aware data. The *process* construct makes a new (sub-)process and defines a new *process context*. Only data variables declared by the current process context are visible. The *scope* construct creates a nested scope and a corresponding *scope context*. A data variable declared by the nearest enclosing scope of the current (sub-)process is visible by the current scope. The environment for process-aware data therefore has a linear structure of process contexts and a nested structure of scope contexts, as below.

$$\overbrace{\{[\underbrace{\ldots, x:v, \ldots}_{\text{current scope}} \underbrace{[\ldots[\ldots]]}_{\text{outer scopes}}]\}}^{\text{visible current process context}} \cdot \overbrace{\{[\ldots[\ldots]]\} \cdot \{[\ldots[\ldots]]\}}^{\text{invisible process contexts}}$$

When a *process* message is interpreted, with Rule (proc) below, a new process context is added to the environment and an *eop* (end-of-process) continuation activity is inserted into the continuation. Process variables (x and y here) declared in the process are visible to the activities in the current process. Later, when the activity (a, the body) of the process is done and the *eop* activity is interpreted, with Rule (eop), the context of the newly finished process is deleted from the new environment.

$$< process(x, y, a) \cdot c, env_0 > \xrightarrow{proc} < a \cdot eop \cdot c, \{x, y\} \cdot env_0 > \qquad \text{(proc)}$$

$$< eop \cdot c, \{x : v_x, y : v_y\} \cdot env_0 > \xrightarrow{eop} < c, env_0 > \qquad \text{(eop)}$$

When a *scope* message is interpreted, the scope context of the current process context is extended such that the new scope context encapsulates the outer scope contexts, thus forming a nested structure of the scope contexts. At the same time, an *eos* (end-of-scope) activity is inserted into the continuation. When the scope terminates and the *eos* activity is interpreted, the context of the terminating scope is deleted from the environment.

$$< scope(x, y, -, a) \cdot c, \{cxt\} \cdot env_0 > \quad \text{(scp\_cxt)}$$
$$\xrightarrow{scp\_cxt} < a \cdot eos(-) \cdot c, \{[x, y, cxt]\} \cdot env_0 >$$
$$< eos(-) \cdot c, \{[x : v_x, y : v_y, cxt]\} \cdot env_0 > \xrightarrow{eos\_cxt} < c, \{cxt\} \cdot env_0 > \quad \text{(eos\_cxt)}$$

In the message representations, we use "−" to suppress some uninteresting details. The interpretation rules (scp_cxt) and (eos_cxt) above are primarily only concerned with the environment part of the message. The interpretation rules for the continuation part will be presented later in Section 4.4.

Process-aware data variables defined in the innermost scope (and thus appear outermost in the environment) or the nearest enclosing scope are visible to the current activities. For example, they can be updated by an *assign* activity.

$$< assign(x, e) \cdot c_2 \cdot eos(-, k) \cdot c_1, \{[\ldots, [x : v_0, \ldots [\ldots]]]\} \cdot \ldots > \quad \text{(asgn)}$$
$$\xrightarrow{asgn} < c_2 \cdot eos(-, assign(x, v_0) \cdot k) \cdot c_1, \{[\ldots, [x : eval(e), \ldots [\ldots]]]\} \cdot \ldots >$$

$eval(e)$ evaluates an expression e (for instance as an XPath expression [57]) in the current process context. The x variable in the nearest enclosing scope is now associated with the new evaluated value. Notice that a compensating *assign* activity is added to the compensation continuation of the scope. If the scope is later rolled back, x will be assigned with its original value v_0. Scope and compensation will be discussed in detail in the following sections.

In what follows, we focus only on the continuation parts of messages and omit the environment parts.

4.3 Commence and termination of orchestration

A client program at site s_c sends message $orch^s(pa, s_c)$ to ask the orchestration agent at site s to orchestrate the process specified in pa. Rule (orch) indicates the commence of an orchestration. The new *eorch* activity in the continuation marks the termination of the orchestration. It also remembers the site of the client.

$$orch^s(pa, s_c) \xrightarrow[s]{orch} pa \cdot eorch^s(s_c) \quad \text{(orch)}$$
$$< eorch(s_c) >_s \xrightarrow[s_c]{eorch} done \quad \text{(eorch)}$$

When all activities of *pa* have finished, *eorch* is the only element in the continuation. Site s sends a reply message to s_c and the orchestration terminates.

4.4 Scopes

In BPEL, scopes are the units of fault handling and recovery. A *scope* activity contains a primary activity as the body of the scope, and optionally, a set of named fault handlers, a set of event handlers, and a compensation handler.

A fault handler is run when some particular fault is thrown from within an unfinished scope.

A compensation handler is run to roll back the effects of a finished scope. Therefore a compensation handler is only provided for a nested scope (i.e., not a top-level scope) and is typically run as part of a fault handler of an enclosing scope.

An event handler handles events asynchronously with the main activities of the scope (such as an alert triggered by a timer).

To keep our discussions to the point, we show below only one fault handler *fh* for handling faults of type f, a single event handler eh, and for a nested scope, a compensation handler ch.

A scope is managed by an OA as its *scope manager*. A scope manager keeps track of some necessary run-time information about the scope, including the current locations of all branches. Choosing the proper OA of a scope manager is a design issue. The default OA of a scope manager is where the scope is created.

The OA of the scope manager is encapsulated in the *eos* activity. Whenever a branch is moving to a new OA, the scope manager is notified. These notification messages are sent in parallel with the normal orchestration messages and thus do not cause any additional delay to process executions.

A *scope* activity creates a scope and installs its event handlers (Rule (scp)). Notice that a scope may have an optional compensation handler ch.

$$scope^{s}(a, f : fh, ch^{?}, eh) \cdot c \xrightarrow[s]{scp} \begin{cases} a \cdot eos(s, id, f : fh, ch^{?}, \bot) \cdot c \\ eh \end{cases} \quad \text{(scp)}$$

When a top-level scope is done (Rule (eos_t)), the event handler eh of the scope is torn down and the execution continues.

$$\left.\begin{array}{l} eos(s, -) \cdot c \\ eh \end{array}\right\} \xrightarrow[s]{eos\_t} c \quad \text{(eos\_t)}$$

When a nested scope terminates, the message is interpreted with Rule (eos).

$$\left.\begin{array}{l} eos(s, -, ch^?, k_2) \cdot c_1 \cdot eos(-, k_1) \cdot c_0 \\ eh \end{array}\right\} \xrightarrow[s]{eos} \begin{cases} c_1 \cdot eos(-, ch \cdot k_1) \cdot c_0 & \text{if } ch \\ c_1 \cdot eos(-, k_2 \cdot k_1) \cdot c_0 & \text{otherwise} \end{cases} \quad \text{(eos)}$$

In the *eos* message, k_2 is the compensation continuation of the terminating scope, k_1 is the compensation continuation of the enclosing (parent) scope, and c_1 is the part of the continuation that represents the remaining work of the parent scope. If the terminating scope has a compensation handler ch, the handler is pushed to the compensation continuation k_1 of its parent *eos* activity. If no compensation handler is given for the scope, the generated compensation continuation k_2 of the scope is pushed to the compensation continuation of the parent scope.

4.5 Structural compositions

Sequential, selective and iterative compositions are pretty straightforward to interpret.

$$sequence(a_1, a_2, \dots) \cdot c \xrightarrow{seq} a_1 \cdot a_2 \cdot \dots \cdot c \quad \text{(seq)}$$

$$if(e, a_t, a_f) \cdot c \xrightarrow{if} \begin{cases} a_t \cdot c & \text{if } eval(e) = true \\ a_f \cdot c & \text{if } eval(e) = false \end{cases} \quad \text{(if)}$$

$$while(e, a) \cdot c \xrightarrow{whl} \begin{cases} a \cdot while(e, a) \cdot c & \text{if } eval(e) = true \\ c & \text{if } eval(e) = false \end{cases} \quad \text{(whl)}$$

A *flow* activity makes a parallel composition so that constituent activities run in parallel. Interpreting a *flow* message with Rule (flw) leads to multiple messages, one for each parallel branch.

$$flow(l, a_1, a_2, \dots) \cdot c_2 \cdot eos(s, -, k) \cdot c_1 \quad \text{(flw)}$$

$$\xrightarrow{flw} \begin{cases} a_1 \cdot join(s, id_j) \cdot c_2 \cdot\ eos(s, -, l : link, join(s, id_k) \cdot k) \cdot\ c_1 \\ a_2 \cdot join(s, id_j) \cdot c_2 \cdot\ eos(s, -, l : link, join(s, id_k) \cdot k) \cdot\ c_1 \\ \dots \end{cases}$$

A *flow* activity may declare dependency links. The declared links are registered in the current *eos* activity. Here we show one link l. We discuss dependency links in Section 4.8.

The parallel branches will later join back into a single one. This is achieved with *join* activities. A *join* activity indicates where the join will occur. Here the branches will join at the location of the current scope manager. Related *join* activities are identified, with id for normal execution and id_k for compensation. In addition, they share a join condition, which must be evaluated to be *true* for the branches to join so that the execution of the process can move on.

In Rule (flw), the join condition is a default *true* expression, meaning that as long as the *join* messages of all branches arrive, execution of the process can move on.

When all the *join* messages are available at the join site s, they are further interpreted with Rule (jon).

$$\left.\begin{array}{l} join(s, id_j) \cdot c_2 \cdot eos(-, k^1 \cdot join(s, id_k) \cdot k) \cdot c_1 \\ join(s, id_j) \cdot c_2 \cdot eos(-, k^2 \cdot join(s, id_k) \cdot k) \cdot c_1 \\ \dots \end{array}\right\} \quad \text{(jon)}$$
$$\xrightarrow{jon} c_2 \cdot eos(-, \mathit{flow}(k^1, k^2, \dots) \cdot k) \cdot c_1$$

Every branch i has generated its compensation continuation k^i during the execution before reaching *join*. When the branches join, the individual compensation continuations are merged into a *flow* activity in the new compensation continuation.

4.6 Service operations

In BPEL, a service operation can be either *request-response* with a *reply* activity or *one-way* without a *reply* message.

A service operation is installed at site s with an *inst* activity.

$$inst(sequence(receive(op), \dots), s) \xrightarrow[s]{install} < receive(op) \cdot \dots >_s \quad \text{(install)}$$

A *receive* message can only be further interpreted together with a matching *invoke* or *send* message. The *receive* message is therefore stored in the pool of pending messages at site s.

We use *invoke* for invoking request-response operations and *send* for invoking one-way operations. Executions of service operations are actually enclosed within implicit scopes. So strictly speaking, *invoke* and *send* are structured activities.

An *invoke* activity may provide fault handlers and a compensation handler. An *invoke* message is interpreted with Rule (inv), when it matches an installed

service operation.

$$\left.\begin{array}{l} invoke(s, op, f : fh, ch^{?}) \cdot c_0 \\ < receive(op) \cdot \ldots \cdot reply(op) >_s \end{array}\right\} \quad \text{(inv)}$$
$$\xrightarrow{inv} \begin{cases} \ldots \cdot eos(id, f : fh, ch^{?}, rethrow) \cdot c_0 \\ < receive(op) \cdot \ldots \cdot reply(op) >_s \end{cases}$$

Interpreting the matching messages leads to two new messages: one for the process running the service operation and the other for re-installing the service operation.

With continuation-passing messaging, the execution can move directly to the subsequent activity without sending a *reply* message to the invoker (so as far as continuation-passing messaging is concerned, the term "request-response" is misleading, because no response message is actually sent to the invoker).

The execution of a service operation is enclosed in a *floating* scope, which does not have its own scope manager. Some of the scope management tasks are delegated to the enclosing non-floating scope manager. If a fault is thrown from within the operation body and the partially executed effects are rolled back, the floating scope has to re-throw the fault to its parent scope.

With a one-way operation, the invoking branch continues its execution after sending the invocation message, and collects the outcome of the operation later with a callback operation. A process calling a one-way service looks like this: $sequence(\ldots, send(s, op), \ldots, receive(\widehat{op}), \ldots)$, where $\widehat{op}$ is the callback operation.

$$\left.\begin{array}{l} send(s, op, f : fh, ch^{?}) \cdot c_i \cdot receive^{\widehat{s}}(\widehat{op}) \cdot\ c_1 \cdot eos(-, k) \cdot c_0 \\ < receive(op) \cdot c_{op} \cdot send(\widehat{op}) >_s \end{array}\right\} \quad \text{(call)}$$
$$\xrightarrow{call} \begin{cases} c_i \cdot receive^{\widehat{s}}(\widehat{op}) \cdot eos(-, receive^{\widehat{s}}(\widehat{op}) \cdot rethrow) \cdot\ c_1 \cdot eos(-, k) \cdot c_0 \\ c_{op} \cdot send(\widehat{s}, \widehat{op}) \cdot eos(-, f : fh, ch^{?}, send(\widehat{s}, \widehat{op})) \\ receive(op) \cdot c_{op} \cdot send(\widehat{op}) \end{cases}$$

In Rule (call), $\widehat{op}$ is the callback operation, $\widehat{s}$ is the site $\widehat{op}$ is expected to be installed, c_i is the continuation representation of the activities at the invoking branch prior to the callback, c_{op} is the continuation representation of the body of the one-way operation, and k is the compensation continuation of the current scope.

The interpretation leads to three new messages that represent the invoking branch, the execution of the one-way operation, and the re-installation of the operation,

With the callback after *op* is successfully done (Rule (callbk)), the two branches are merged into one. The recovery plans of the two branches are composed into

a *flow* activity. Similar to the request-response case, if no compensation handler is provided during the invocation, the generated compensation continuation k_{op} will be used for compensation.

$$\left.\begin{array}{l} < receive(\widehat{op}) \cdot eos(-, k_i \cdot receive^{\widehat{s}}(\widehat{op}) \cdot rethrow) \cdot\ c_1 \cdot eos(-, k) \cdot c_0 >_{\widehat{s}} \\ send(\widehat{s}, \widehat{op}) \cdot\ eos(-, ch^?, k_{op} \cdot send(\widehat{s}, \widehat{op})) \end{array}\right\}$$
$$\xrightarrow{callbk} \begin{cases} c_1 \cdot eos(-, flow(k_i, ch) \cdot k) \cdot c_0 & \text{if } ch \\ c_1 \cdot eos(-, flow(k_i, k_{op}) \cdot k) \cdot c_0 & \text{otherwise} \end{cases} \quad \text{(callbk)}$$

In the messages as the result of Rule (call), the the compensation continuations include the same callback as in the normal continuations. If operation *op* fails and rolls back, this callback tells the invoker about the fault. The invoker interprets this *send* message with Rule (callbk_k_op) and throws a fault.

$$\left.\begin{array}{l} < receive(\widehat{op}) \ \cdot\ eos(-, k_i \cdot receive(\widehat{op}) \cdot rethrow) \\ \qquad \cdot c_1 \cdot eos(-, k) \cdot c_0 >_{\widehat{s}} \\ send(\widehat{s}, \widehat{op}) \cdot\ eosk(-) \end{array}\right\} \quad \text{(callbk\_k\_op)}$$
$$\xrightarrow{callbk\_k} throw \cdot eos(-, k_i \cdot receive(\widehat{op}) \cdot rethrow) \cdot\ c_1 \cdot eos(-, k) \cdot c_0$$

Similarly, if the invoking branch fails and rolls back, it will also wait for the same callback (Rule (callbk_k_i)).

$$\left.\begin{array}{l} < receive(\widehat{op}) \ \cdot\ eosk(-, rethrow) \cdot\ c_1 \cdot eos(-, k) \cdot c_0 >_{\widehat{s}} \\ send(\widehat{s}, \widehat{op}) \cdot\ eos(-, ch^?, k_{op} \cdot send(\widehat{s}, \widehat{op})) \end{array}\right\} \quad \text{(callbk\_k\_i)}$$
$$\xrightarrow{callbk\_k} throw \cdot\ c_1 \cdot eos(-, k) \cdot c_0$$

If the invoker of a one-way operation does not expect any result of the operation, the invoking branch and the operation branch join before the end of the enclosing scope, as stated by Rule (snd).

$$\left.\begin{array}{l} send(s, op, f : fh, ch,) \cdot c_1 \cdot eos(s_0, -, k) \cdot c_0 \\ < receive(op) \cdot c_{op} >_s \end{array}\right\} \quad \text{(snd)}$$
$$\xrightarrow{snd} \begin{cases} c_1 \cdot join(s_0, id_1) \cdot eos(s_0, -, join(s_0, id_2) \cdot k) \cdot c_0 \\ c_{op} \cdot eos(id_3, f : fh, ch, rethrow) \\ \quad \cdot join(s_0, id_1) \cdot eos(s_0, -, join(s_0, id_2) \cdot k) \cdot c_0 \\ < receive(op) \cdot c_{op} >_s \end{cases}$$

Notice that all the five rules above, i.e. Rules (call), (callbk), (cakkbk_k_op), (callbk_k_i) and (snd), are used to interpret a pair of *receive* and *send* messages.

In these cases, the head activities of the continuations alone are not sufficient to determine which rule to apply. We have to look further for other related activities in the continuations, for example, whether a *receive* or *send* activity is followed immediately with an *eos* or *eosk* activity.

4.7 Fault handling

Every scope is associated with a number of fault handlers that are first provided in the the *scope* activity and later encapsulated in the *eos* activity. When a fault f is thrown within the scope, the corresponding fault handler is executed. If the *scope* activity does not have a fault handler for f, a default fault handler will run, which is typically $\mathit{sequence}(\mathit{compensate}, \mathit{rethrow})$. That is, the default fault handler rolls back the finished activities of the scope and re-throws the caught exception to the parent scope.

Because the fault handler is encapsulated in the *eos* activity of the continuation, fault handling can start immediately when a fault is thrown. However, a scope may have several parallel branches concurrently, the fault must also be propagated to the other branches. This is achieved through the *notify* activity.

In Rule (thw) below, assume that Branch 1 throws a fault f from within a nested scope. Moreover, suppose that the fault handler consists of three parts: fh_1, *compensate* and fh_2.

$$\mathit{throw}(f) \cdot c_1^1 \cdot \mathit{eos}(s, id, f : \mathit{fh}_1 \cdot \mathit{compensate} \cdot \mathit{fh}_2, ch, k^1) \cdot c_0 \qquad \text{(thw)}$$
$$\xrightarrow{thw} \begin{cases} \mathit{notify}(s, id, f) \\ \mathit{fh}_1 \cdot k^1 \cdot \mathit{fh}_2 \cdot \mathit{eosk}(s, id) \cdot c_0 \end{cases}$$

In the *throw* continuation, c_1^1 represents the remaining work of the scope and k^1 represents the current compensation continuation of the scope. Catching the fault leads to two messages, one for notifying the other branches and the other for handling the fault.

Handling the fault is achieved through the following changes in the continuation: (1) the remaining continuation of Branch 1 within the current scope, c_1^1, is discarded, (2) the *eos* activity is replaced with *eosk* (end-of-scope in compensation), and (3) the fault handler is applied in the new continuation. Since in BPEL, a compensation itself cannot be compensated for, *eosk* does not contain a compensation continuation. The *compensate* activity in the fault handler is replaced with the compensation continuation k^1 of Branch 1 for rolling back the effects that have been achieved by the execution so far.

A fault is notified to other branches via the scope manager with the *notify* message. That is, the site throwing the fault sends a *notify* message to the

scope manager which then forwards the messages to the current sites of the other branches. These branches, once receiving the *notify* messages, start handling the fault, similar to the fault-throwing site (Rule (nty)).

$$\left.\begin{array}{l}\text{for all branches } i = 2, \ldots, n \\ c_2^i \cdot eos(s_1^i, -, k_1^i) \cdot c_1^i \cdot eos(s, id, f : fh_1 \cdot compensate \cdot fh_2, k^i) \cdot c_0 \\ eh^i \\ eh \\ notify(s, id, f)\end{array}\right\} \quad \text{(nty)}$$

$$\xrightarrow{nty} \begin{cases}\text{for all branches } i = 2, \ldots, n \\ k_1^i \cdot k^i \cdot fh_2 \cdot eosk(s, id) \cdot c_0\end{cases}$$

Notice that these branches may have entered new scopes and created new event handlers. In Rule (nty), we assume that all these branches have entered a nested scope. The recovery thus must also rollback these child scopes.

The executed effects of all branches of the faulty scope are rolled back in parallel branches represented by the compensation continuations k^i $(i = 1, \ldots, n)$. These recovery branches eventually join into a single one. For example, if the parallel branches were created with *flow*, all recovery branches will reach $join(s, id_k)$ (see Rule (flw)). Consequently, the fh_1 and fh_2 parts of the fault handler are run only once (fh_1 by Branch 1 and fh_2 by the OA that joins the branches).

A *rethrow* from a fault handler is captured by the parent scope. Its handling is similar to a *throw*.

4.8 Dependency links

In a BPEL process, *dependency links* enforce additional precedence dependencies among activities. An activity that is targets of some links can not start before the source activities of these links finish. Figure 4.2 shows and example of a dependency link.

A link is first declared by a *flow* activity and then attached to some activities in different branches of the flow. An activity attached with links takes the form (ba, ln^+, e), where ln^+ is the set of links and e is a join condition on the status of the target links. An activity can only be further interpreted when the join condition evaluates to be *true*.

Because branches may create new scopes, links may cross scope boundaries. If a source activity of a link is not declared inside an enclosing scope, the link is said to be *leaving* the scope. Similarly, if a target activity of a link is not declared inside an enclosing scope, the link is said to be *entering* the scope.

$$
\begin{array}{l}
fork(\\
\quad sequence(\\
\quad\quad invoke(S_a, a, l_{ad} : source, true), \\
\quad\quad invoke(S_b, b)), \\
\quad sequence(\\
\quad\quad invoke(S_c, c), \\
\quad\quad invoke(S_d, d, l_{ad} : target, true)), \\
\quad l_{ad})
\end{array}
$$

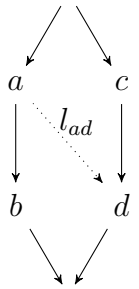


Figure 4.2: A dependency link

Because incoming and outgoing links may influence the behavior of a scope, such as the compensation ordering, an *eos* activity keeps track of the links declared in this scope as well as the incoming and outgoing links.

$$eos(-, l_s : link, l_{out} : outgoing, l_{in} : incoming)$$

For the sake of readability, in what follows, we will not show the links in *eos* activities.

A source activity is interpreted with Rule (src).

$$(a, l^s : source, e) \cdot c \xrightarrow{src} a \cdot lnk^s(l, e) \cdot c \tag{src}$$

If the link is not declared in the current scope, it will be registered in the *eos* as an outgoing link. To enforce the execution precedence, a *lnk* activity is inserted after the source activity.

A *lnk* message is interpreted with Rule (lnk).

$$lnk^s(l, e_s) \cdot c_1 \cdot eos(-, k) \cdot c_0 \xrightarrow{lnk} \begin{cases} c_1 \cdot eos(-, (\tau, \bar{l} : target, true) \cdot k) \cdot c_0 \\ lnk^s(l, e) \end{cases} \tag{lnk}$$

Interpreting a *lnk* message leads to a message with the single *lnk* activity to the OA of the corresponding target activity.

A link $\bar{l}$ is introduced in the compensation continuations to enforce the right compensation order (τ is an empty activity). That is, compensating the effect of l's target activity must take effect prior to compensating the effect of l's source activity. In the branch of l's source activity, an empty target activity with link $\bar{l}$ is now pushed to the new compensation continuation.

A target activity is interpreted, with Rule (tgt), when all its dependent *lnk*

messages arrive. Here we show an activity with only one target link.

$$\left.\begin{array}{l}(a, l : \mathit{target}, e_2) \cdot c_1 \cdot \mathit{eos}(-, k) \cdot c_0 \\ \mathit{lnk}(l, e_1)\end{array}\right\} \xrightarrow{tgt} \begin{array}{l}\mathit{if}(\mathit{eval}(e_1 \wedge e_2), \\ \quad a, \\ \quad \mathit{if}(\mathit{suppressJoinFailure}, \\ \quad\quad \tau, \\ \quad\quad \mathit{throw}(\mathit{joinFailure}))) \\ \cdot\, c_1 \cdot \mathit{eos}(-, \mathit{lnk}(\bar{l}, \mathit{true}) \cdot k) \cdot c_0\end{array} \quad \text{(tgt)}$$

If the join condition evaluates to be *true*, the target activity a will run. If the join condition evaluates to be *false* and the *suppressJoinFailure* flag is on, the target activity a is skipped. If the join condition evaluates to be *false* and the *suppressJoinFailure* flag is off, a $joinFailure$ fault is thrown.

A $\mathit{lnk}(\bar{l}, \mathit{true})$ activity is pushed to the compensation continuation so that compensation of l's source activity follows the compensation of l's target activity through link $\bar{l}$.

For dependency links, there is a special issue, known as the *dead-path* issue. If a source activity is part of a selective composition and is not run, the target activity may wait forever. One solution to this issue is to *falsify* all source activities inside a selective constituent that is not run. For example, Rule (if_fsy) below falsifies the activity that is not run.

$$\mathit{if}(b, a_t, a_f) \cdot c \xrightarrow{\mathit{if\_fsy}} \begin{cases} \mathit{falsifylinks}(a_f) \cdot a_t \cdot c & \text{if } \mathit{eval}(b) = \mathit{true} \\ \mathit{falsifylinks}(a_t) \cdot a_f \cdot c & \text{if } \mathit{eval}(b) = \mathit{false} \end{cases} \quad \text{(if\_fsy)}$$

$$\mathit{falsifylinks}(\ldots (a, l : \mathit{source}, e) \ldots) \cdot c \xrightarrow{\mathit{fsy}} \begin{cases} \mathit{lnk}(l, \mathit{false}) \\ c \end{cases} \quad \text{(fsy)}$$

4.9 Example

We run the example process in Figure 4.3 to illustrate concretely how CPM works.

The process consists of a scope. The body of the scope is a *flow* activity of two parallel branches. The first branch invokes two service operations A and B in sequence. The second branch is enclosed in a nested scope. It invokes service operations C and D in sequence. The nested scope has a compensation handler that compensates for the completed effects of C and D with operation F. Each of the scopes has a default fault handler with a single *compensate* activity that

$$
\begin{aligned}
process(scope(\ flow(\ sequence(\ & invoke(S_A, A, invoke(S_A, \overline{A})),\\
& invoke(S_B, B, invoke(S_B, \overline{B}))),\\
& scope(\ sequence(\ send(S_C, C),\\
& \qquad \vdots\\
& \qquad receive(\widehat{C}),\\
& \qquad invoke(S_D, D, invoke(S_D, \overline{D}))),\\
& any : compensate,\\
& invoke(S_F, F))),\\
& any : compensate))
\end{aligned}
$$

Figure 4.3: Example process

rolls back the partial execution of the scope. None of the scopes have any event handler.

In what follows, we refer to remote messages by numbers and local messages by numbers prefixed with the site identifier. For example, message 1 is a remote message and message $P1$ is a local message at site O_P.

4.9.1 Service installation

In order to illustrate features for service operations, we assume that all service providers in our example are also orchestration agents.

Suppose that service operations A, B, ..., and their compensation operations $\overline{A}$, $\overline{B}$, ..., are installed at sites S_A, S_B, ... respectively. Suppose further that only operation C is one-way; all the other operations are request-response.

Service operation A looks like this:

$$sequence(receive(A), invoke(S_E, E, invoke(S_E, \overline{E})), reply(A))$$

That is, service operation A invokes service operation E from within its body. Installing A at site S_A results in a *receive* message $A1$ stored in the pool of pending messages at S_A.

$$
\begin{aligned}
& inst(sequence(receive(A), -))\\
& \xrightarrow[S_A]{inst} < receive(A) \cdot invoke^{S_E}(S_E, E, invoke^{S_E}(S_E, \overline{E})) \cdot reply(A) >_{S_A} \qquad \text{(A1)}
\end{aligned}
$$

The other service operations are similarly installed. The messages for the installed operations $A, B, \ldots$ and $\overline{A}, \overline{B}, \ldots$ are $A1, B1, \ldots$ and $A2, B2, \ldots$. They are stored in the pools of pending messages at sites S_A–S_E, after the installation messages are interpreted. The one-way service operation C provides results on callback operation $\widehat{C}$.

$$< receive(A) \cdot \ldots \cdot reply(A) >_{S_A} \qquad (A1,\ B1,\ \ldots)$$
$$< receive(\overline{A}) \cdot \ldots \cdot reply(\overline{A}) >_{S_A} \qquad (A2,\ B2,\ \ldots)$$
$$< receive(C) \cdot \ldots \cdot send(\widehat{C}) >_{S_C} \qquad (C1)$$

4.9.2 Successful execution

Figure 4.4 shows the remote messages for the orchestration of a successful execution of the process.

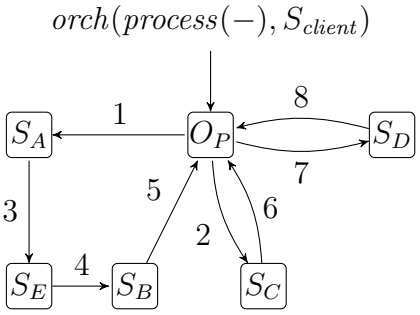


Figure 4.4: Orchestration messages for a successful execution

A process instance is created when an *orch* message arrives at O_P.

$$< orch^{O_P}(process(-), S_{client}) >_{S_{client}} \xrightarrow[O_P]{orch} process(-) \cdot eorch^{O_P}(S_{client}) \qquad (P1)$$

The initial process instance is represented as message $P1$. Figure 4.5 shows how orchestration agents are assigned to specific activities.

After a series of interpretation steps, the message of the process instance is turned into a *flow* message $P2$.

$$\begin{aligned} P1 &\xrightarrow{proc} scope(-) \cdot eop \cdot eorch(S_{client}) \\ &\xrightarrow{scp} flow(-) \cdot eos(O_P, 1, any : compensate, \bot) \cdot c_{eop} \end{aligned} \qquad (P2)$$

where

$$c_{eop} = eop \cdot eorch(S_{client}) \qquad (c_{eop})$$

$$
\begin{aligned}
&process(scope^{O_P}(\ flow(\ sequence(\ invoke^{S_A}(S_A, A, invoke^{S_A}(S_A, \overline{A})),\\
&\qquad\qquad\qquad\qquad invoke^{S_B}(S_B, B, invoke^{S_B}(S_B, \overline{B}))),\\
&\qquad\qquad scope^{O_P}(\ sequence(\ send(S_C, C),\\
&\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad\qquad\qquad receive^{OP}(\widehat{C}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad invoke^{S_D}(S_D, D, invoke^{S_D}(S_D, \overline{D}))),\\
&\qquad\qquad\qquad\qquad any : compensate,\\
&\qquad\qquad\qquad\qquad invoke^{O_P}(S_F, F))),\\
&\qquad\qquad any : compensate))
\end{aligned}
$$

Figure 4.5: Process in message (P1)

Interpreting $P2$ leads to two messages, 1 and 2, representing two parellel branches, that are sent to S_A and S_C. Notice the two *join* activities in each of the messages, one for the normal execution and one for rollbacks.

$$
P2 \xrightarrow{flw} \begin{cases} \xrightarrow{seq} invoke(S_A, A, -) \cdot invoke(S_B, B, -) \cdot\ c^2_{join} & (1)\\ \xrightarrow{scp,seq} send(S_C, C) \cdot \ldots \cdot receive(\widehat{C}) \cdot\ invoke(S_D, D, -) & (2)\\ \qquad \cdot eos(O_P, 4, any : compensate, invoke(S_F, F), \bot) \cdot\ c^2_{join} & \end{cases}
$$

where

$$
c^2_{join} = \ join(O_P, 2) \cdot\ eos(O_P, 1, any : compensate, join(O_P, 3)) \cdot c_{eop} \qquad (c^2_{join})
$$

The first branch invokes operation A that in turn invokes operation E with message 3.

$$
\left.\begin{matrix} 1 \\ A1 \end{matrix}\right\} \xrightarrow{inv} \begin{cases} invoke(S_E, E) \cdot eos(5, invoke(S_A, \overline{A}), rethrow) & \\ \quad \cdot invoke(S_B, B, -) \cdot c^2_{join} & (3)\\ A1 & \end{cases}
$$

After operation E is done, the invocation of its compensation operation $\overline{E}$ is pushed to the compensation continuation of the $eos(5, -)$ activity of the enclosing

scope, i.e., the floating scope of operation A.

$$\begin{aligned} 3 \xrightarrow{inv,\ldots,eos} & \; eos(5, invoke(S_A, \overline{A}), invoke(S_E, \overline{E}) \cdot rethrow) \\ & \cdot invoke(S_B, B, -) \cdot c_{join} \\ \xrightarrow{eos} & \; invoke(S_B, B, -) \cdot join(O_P, 2) \\ & \cdot \; eos(O_P, 1, any : compensate, invoke(S_A, \overline{A}) \cdot join(O_P, 3)) \\ & \cdot c_{eop} \end{aligned} \tag{4}$$

Further, when the $eos(5, -)$ of operation A is interpreted, $invoke(\overline{A})$ for compensating A is pushed to the *eos* activity of the top-level scope. This branch moves further on with an *invoke* message 4 directly from S_E to S_B. Note that there is no reply message to S_A when operation E is done, or to O_P when operation A is done.

Notice that $invoke(\overline{E})$ in the compensation continuation of the $eos(5, -)$ allows $\overline{E}$ to reverse the effect of E when there is a fault and operation A has *not* finished. When the execution of A has finished, the compensation handle $invoke(\overline{A})$ is inserted in $eos(1, -)$ of the enclosing scope and the compensation continuation in $eos(5, -)$ is discarded.

When operation B is done, a *join* message 5 is sent back to O_P. The compensation continuation of the branch is updated accordingly. If the *join* message of the other branch has not arrived yet, message 5 is stored at O_P as a pending message.

$$\begin{aligned} 4 \xrightarrow{inv,\ldots,eos} & \; join(O_P, 2) \\ & \cdot eos(O_P, 1, any : compensate, \\ & \qquad invoke(S_B, \overline{B}) \cdot invoke(S_A, \overline{A}) \cdot \; join(O_P, 3)) \\ & \cdot c_{eop} \end{aligned} \tag{5}$$

The second branch is enclosed in a nested scope, managed also at O_P. In the scope, the one-way operation C is invoked. This in turn leads to two new branches (in addition to $C1$ that re-installes C), one continuing execution at the invoking site O_P and the other running the one-way operation C.

$$\left.\begin{matrix} 2 \\ C1 \end{matrix}\right\} \xrightarrow{call} \begin{cases} \xrightarrow{\cdots} < receive(\widehat{C}) \cdot eos(7, k_i \cdot rethrow) \cdot \; invoke(S_D, D, -) & (P3) \\ \qquad \cdot eos(O_P, 4, any : compensate, invoke(S_F, F), \bot) \\ \qquad \cdot c^2_{join} >_{O_P} \\ \xrightarrow{\cdots} send(O_P, \widehat{C}) \cdot eos(O_P, 6, k_C \cdot send(O_P, \widehat{C})) & (6) \\ C1 \end{cases}$$

After doing some work at O_P, the invoking branch waits for the result of C with callback $\widehat{C}$. The *receive* message for the callback is stored in the pool of pending messages at O_P. After receiving the result from the callback message 6, this branch moves further on to operation D with message 7. k_i and k_C are the compensation continuations generated during the executions of the invoking branch and the one-way operation C, respectively. The nested scope is terminated with the $eos(4,-)$ message 8 back to O_P. Interpreting the *eos* message leads to a local *join* message $P4$.

$$
\begin{aligned}
&\left.\begin{matrix} P3 \\ 6 \end{matrix}\right\} \xrightarrow{callbk} \\
&\quad invoke(S_D, D, -) && (7)\\
&\qquad \cdot\, eos(O_P, 4, any : compensate, invoke(S_F, F), flow(k_i, k_C)) \\
&\qquad \cdot\, c^2_{join} \\
&\xrightarrow{inv,\ldots,eos} eos(O_P, 4, any : compensate, invoke(S_F, F), && (8)\\
&\qquad\qquad\qquad invoke(S_D, \overline{D}) \cdot flow(k_i, k_C)) \\
&\qquad\qquad \cdot\, c^2_{join} \\
&\xrightarrow{eos} join(O_P, 2) && (P4)\\
&\qquad \cdot\, eos(O_P, 1, any : compensate, invoke(S_F, F) \cdot join(O_P, 3)) \\
&\qquad \cdot\, c_{eop}
\end{aligned}
$$

When the *join* messages of both branches, 5 and $P4$, are available at O_P, the process moves on and finally sends a *reply* message back to the client.

$$
\begin{aligned}
&\left.\begin{matrix} 5 \\ P4 \end{matrix}\right\} \xrightarrow{jon} \\
&\quad eos(O_P, 1, any : compensate, \\
&\qquad\quad flow(invoke(S_B, \overline{B}) \cdot invoke(S_A, \overline{A}), invoke(S_F, F))) && (P5)\\
&\qquad \cdot\, c_{eop} \\
&\xrightarrow{eos} eop \cdot eorch(S_{client}) \\
&\xrightarrow{eop} eorch(S_{client}) \\
&\xrightarrow[S_{client}]{eorch} done
\end{aligned}
$$

Notice that the $eos(1,-)$ activity of message P5 contains the rollback plan of the entire process.

4.9.3 Rollback after a fault

Suppose now that service operation B throws a fault. The latest remote message that S_B has received is message 4 in Figure 4.4. Figure 4.6 shows the remote messages for fault propagation and process recovery.

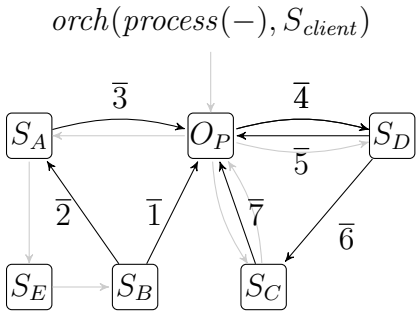


Figure 4.6: Orchestration messages for a rollback

The process state at the faulty branch is represented by message 4. The fault is caught by the fault handler in the *eos* activity of the top scope.

$$4 \xrightarrow{\ldots,thw} \begin{cases} notify(O_P, 1, any) & (\overline{1}, \overline{4}) \\ invoke(S_A, \overline{A}) \cdot c^3_{join} & (\overline{2}) \end{cases}$$

where

$$c^3_{join} = join(O_P, 3) \cdot eosk(O_P, 1) \cdot c_{eop} \qquad (c^3_{join})$$

Catching the fault leads to two messages, $\overline{1}$ for fault propagation and $\overline{2}$ for handling the fault at the faulty branch.

The scope manager at O_P will further propagate the fault, with message $\overline{4}$, to the other branch, currently at S_D, represented by message 7.

The fault handler of the faulty branch consists of a single *compensate* activity. So compensation of the branch starts by applying the compensation continuation encapsulated in the *eos* activity. This leads to message $\overline{2}$ that invokes the compensation operation $\overline{A}$, followed by message $\overline{3}$ that joins with the other recovery branch at O_P.

$$\overline{2} \xrightarrow{inv,\ldots,eos} c^3_{join} \qquad (\overline{3})$$

The other branch, upon receiving the notification message $\overline{4}$, starts handling the fault and rolls back the execution that has been done so far. Because this branch has a nested scope, rolling back the top scope includes the rollback of the

nested scope, too.

$$\left.\begin{matrix}\overline{4}\\7\end{matrix}\right\} \xrightarrow{nty} flow(k_i, k_C) \cdot c^3_{join}$$

$$\xrightarrow{flw} \begin{cases} k_i \cdot join(O_P, 8) \cdot c^3_{join} & (\overline{5}) \\ k_C \cdot join(O_P, 8) \cdot c^3_{join} & (\overline{6}) \end{cases}$$

Since operation C is one-way, the compensation happens in two branches, message $\overline{5}$ for rolling back the effects at the invoking site O_P with the generated compensation continuation k_i, and message $\overline{6}$ for rolling back the effects of C itself. Since no compensation operation was provided when C was invoked, the compensation continuation k_C generated during the execution of C becomes the recovery plan. These two branches then join at O_P.

$$\left.\begin{matrix}\overline{5} \dashrightarrow join(O_P, 8) \cdot c^3_{join} \\ \overline{6} \dashrightarrow join(O_P, 8) \cdot c^3_{join} \quad (\overline{7})\end{matrix}\right\} \xrightarrow{jon} c^3_{join} \qquad (\overline{P1})$$

Finally, the two recovery branches of the top-level scope join.

$$\left.\begin{matrix}\overline{3}\\ \overline{P1}\end{matrix}\right\} \xrightarrow{jon,eosk} c_{eop}$$

4.10 Related work

Services orchestration is typically carried out by central engines, although this is generally regarded as neither scalable nor reliable [1]. Our performance study described later in Chapter 6 shows that even with techniques like replication or pooling [58], centralized approaches cannot achieve the scalability of decentralized ones. Moreover, for the ever-growing Internet-scale applications, adoption of a central engine is hardly possible.

With instantiation-based decentralization, a process is instantiated before execution (for example, [8, 22, 55, 56, 68]). Resources and control are allocated in the distributed environment during instantiation. These approaches are more suitable for enterprise applications where the overhead of instantiation is apportioned to the repeated executions. For more dynamically composed services, messaging-based approaches incur less run-time overhead for services orchestration. A more serious problem with instantiation-based approach for open web applications is that it is practically unacceptable to allocate resources for the parts (often provided by other web applications) that are actually not executed, such as some of the alternative paths or when a process rolls back at an early stage.

Messaging-based approaches (for example, [11, 52, 67, 82]) replace the overhead of instantiation with larger (and sometimes more) messages. In the earlier messaging-based approaches, parts of the static specification of the process, represented as mobile code, is carried in messages.

In all the aforementioned decentralized approaches, whether instantiation-based or messaging-based, decentralization is based on the static structure of the process. There lacks a general mechanism for the features that require dynamic run-time information, such as fault handling and recovery. Furthermore, they tend to have limited adaptability at run time, because the control is mostly already in place before the execution started. Typically, tasks like fault handling and recovery are delegated to a single dedicated site [8, 14, 22, 52, 82]. They are therefore subject to similar limitations of a central engine. With our approach, a scope manager plays a similar role of the dedicated site. However, the number and locations of scope managers are not fixed. Moreover, fault handlers and dynamically generated recovery plans are encapsulated in messages. Consequently, a fault is caught anywhere it is thrown, rather than be propagated to a pre-selected site.

Services orchestration can also be supported with rule-based or event-driven approaches. PADRES [44, 45] is a decentralized event-driven system for services orchestration. It adopts a publish/subscribe approach, where a message broker takes care of the interdependencies among activities specified in composite subscriptions. PADRES supports content-based routing in a network of overlay brokers, resembling a pool of engines. [44] shows a case with 5 overlay brokers. [45] extends to 30 brokers with 20 publishers and 30 subscribers.

INCA [5] is a rule-based system with some properties similar to our approach. A message carries rules and a log. The rules and the log play the roles of normal and compensation continuations of our approach. Besides the principle difference between the approaches (rule-based versus continuation-passing), there are some subtle differences in what can be achieved. With INCA, a site conducts process execution using both the rules carried in messages and pre-installed local rules. Thus INCA can be regarded as a hybrid of instantiation-based and messaging-based approaches.

The concept of continuation is widely used in the programming language community, particularly of functional programming languages. We refer interested readers to [27] for an excellent introduction of the concept.

Micro-workflow [48] provides a software framework that uses continuations for workflow enactment, similar to our work. The purpose of the framework is to support separation of the control concern from the other concerns during software development. There is thus limited support for distribution: remote workflows are enacted with synchronous remote procedure calls. There is no support for recovery.

Our work is built on CEK^T [41] and PCKS [54] machines, which are extensions to the CEK machine [24] (a variant of the SECD machine [42]) for interpreting functional programs using continuations. The CEK^T machine supports asynchronous execution of distributed programs [41]. Upon invocation of a remote procedure, a continuation is passed to the agent, which, after executing the procedure, applies the continuation instead of returning the control back to the caller. Our handling of invocations of request-response operations follows this approach. CEK^T, however, supports only one (distributed) thread of control. The PCKS machine supports parallel executions of functional programs in a shared-memory environment [54].

Success and failure continuations have been applied in execution of logical programs [75] and specification of denotational semantics of stateflows [34]. There, the use of success and failure continuations is similar to the treatment of an *if* activity (to deal with the cases where the condition evaluates to true or false). To our knowledge, we are the first to use compensation continuations for recovery. We are not aware of any published work on enforcing correct compensation order with explicit control dependencies in decentralized approaches.

4.11 Summary

In this chapter, we worked out the details of continuation-passing messaging (CPM), including the construction of CPM message, the rules for controlling the execution flows and fault handling. One special property of CPM, as opposed to approaches based on remote procedure call (RPC), is that after the execution of a service operation, the execution can directly move to the next service site without sending a *reply* message back to its invoker.

As the relevant fault handler is encapsulated in the *eos* activity of the continuation, fault handling can start immediately when a fault is thrown. In case of multiple parallel branches, the site throwing the fault also sends a *notify* message to the scope manager and later the scope manager notifies other branches about the fault.

Section 4.9 illustrate how CPM works with an example. It shows both a normal execution and a scenario where a service operation fails and requires rollback.

Finally, this chapter surveyed related work, focusing on related decentralized approaches. The subtle difference between earlier approaches and our approach is that we utilize dynamic run-time information for fault handling and recovery purposes. Thus we eliminate the necessities of any designated site for exception or fault handling.

Chapter 5

Replicated CPM

In a wide open distributed environment, failures may occur. In our system model, failures may occur either at SPs or at OAs.

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 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. As discussed in Chapter 4 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. This chapter presents the replicated CPM that enhances the availability of the orchestration when the OAs are subject to such unavailability.

5.1 Overview

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.

Consider the example service-base application (SA) in Figure 2.2 and an orchestration of its execution in Figure 3.3. When OA A_c sends orchestration message 5 to OA A_d, A_c has the latest state information about the current branch.

If A_c does not discard this information, A_c can serve as a backup of A_d. If A_d crashes during the execution of service d, A_c could take over the role of A_d and resume the orchestration.

In addition, when OA A_c sends orchestration message 5 to OA A_d, it also sends a scope message to the current scope manager A_p. If in addition to the scope message, A_c also sends A_p an update of the orchestration state information, A_p can serve as an additional backup of both A_c and A_d.

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 its 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.

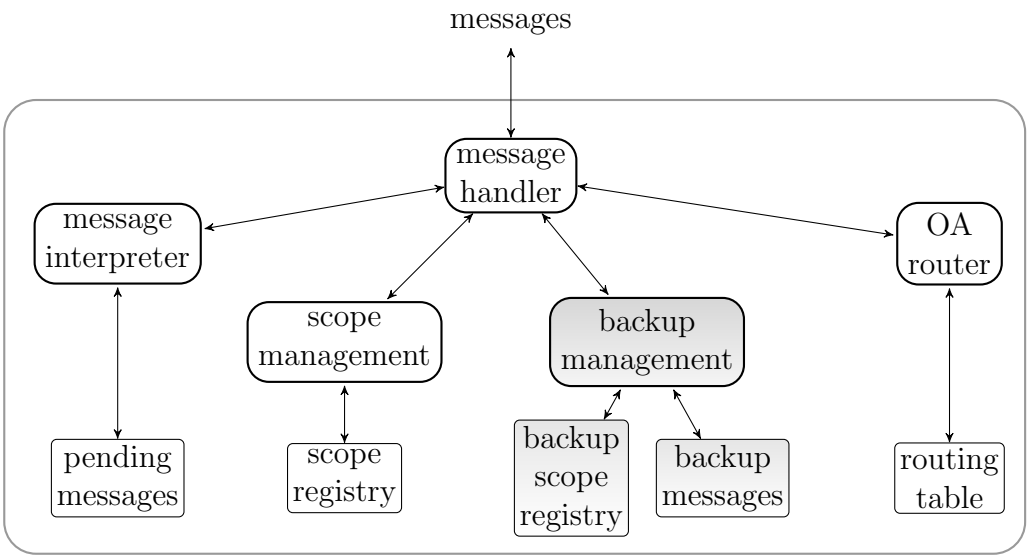


Figure 5.1: Extended Structure of an Orchestration Agent

Figure 5.1 shows the structure of an OA extended with replicated CPM support. It extends the structure of the basic OA in Figure 3.2 with additional components for backup management.

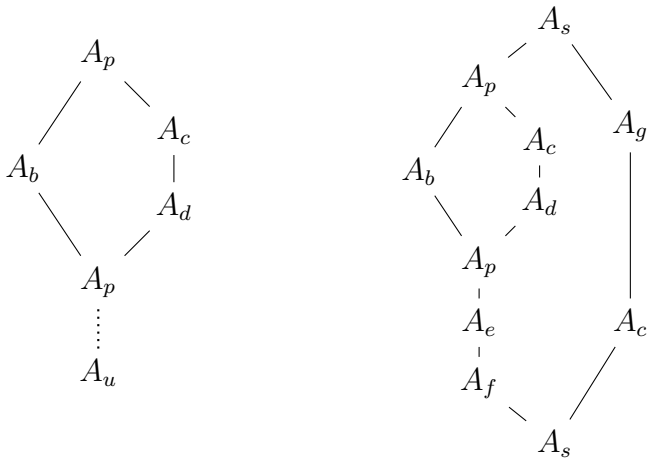
5.2 Selection of backup OAs

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.



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

Figure 5.2: OA graph for backup selection

Figure 5.2 shows two OAGs. The OAG to the left consists of OAs assigned to an orchestration of the example SA p of Figure 2.2. The OAG to the right is an extension s with more service operations. 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 Figure 5.2, 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. For p, the number of assigned OAs is 4. An OAG with degree 5 can be obtained by appending one more OA to the youngest node A_p, as A_u in Figure 5.2. 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.

   (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.

These rules are based on the lifetime and freshness of the run-time and management data maintained by the OAs. The OAs of the enclosing scopes of a given activity have higher priority (Rule S1), because the enclosing scopes maintains more status data related to the activity and live longer. The OAs closer to A in the OAG have higher priorities (Rules S1.a and S2.b), because they have fresher status data relevant to the activities assigned to A. Ascendant OAs (Rule S2.a)

or OAs closer to the scope manager (Rule S2.d) have higher priorities , because they have already orchestrated some activities and thus know more details about what has happened. In addition, the activities for the descendant OAs may never happen in case of exception events. OAs in branches with more activities have higher priorities (Rule S2.c), because there are more message exchanges there and the status information is more likely to be fresh.

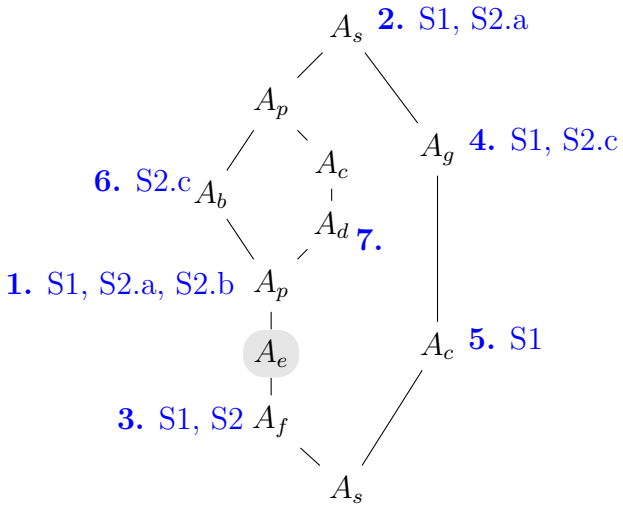


Figure 5.3: Backups of A_e

Figure 5.3 shows the priorities of the backups for A_e in Figure 5.2 and which rules are applied. Suppose A_s is the scope manager of the top-level scope and A_p is the scope manager of the nested scope. The OAs A_s, A_p, A_f, A_g and A_c are assigned for activities in the same scope as A_e. They have higher priorities than A_b and A_d that are only assigned for activities in the nested scope (Rule S1). A_s, A_p and A_f are in the same branch as A_e. They have higher priorities than A_g and A_c (Rule S2). A_s and A_p are ascendants of A_e (Rule S2.a). A_p is closer to A_e than A_s (Rule S2.b). Therefore according to Rules S1, S2.a and S2.b, A_p has the highest priority. Figure 5.3 shows the rule for the other backup agent.

The following table shows the lists of backup candidates (with length 4 for p and 7 for s) of the OAs for the two OAGs in Figure 5.2.

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

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.

5.3 Normal execution

The RCPM mechanism extends CPM as described in Chapter 4 with the following tasks:

- it keeps the backup OAs updated about the lasted orchestration states, and
- it informs the backup OAs to purge the backup states when they are no longer needed.

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 Figure 5.4, 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 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 there exists $k(0 < k \leq n)$ such that $n_0^1 = n_0^2, \ldots, n_{k-1}^1 = n_{k-1}^2$ and

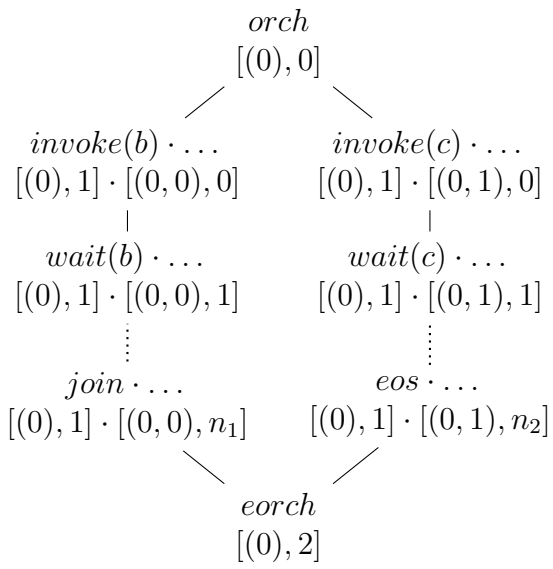


Figure 5.4: Message timestamps

$n_k^1 < n_k^2$. That is, the m^1 and m^2 are for activities in the same branch $b_k^1 (= b_k^2)$, and m^2 has a larger sequence number n_k^2. Messages m^1 and m^2 are *concurrent*, denoted $m^1 \parallel m^2$, if $n_0^1 = n_0^2, \ldots, n_i^1 = n_i^2$. That is, m^1 and m^2 are for activities in two different branches b_{i+1}^1 and b_{i+1}^2.

Suppose OA A received a messages c_0, has done some local interpretations and 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 $c.\mathcal{S}^+ \cup \mathcal{G}$ 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 pending message pool, it also sends the delta of the message c 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 OA manager (Figure 5.1). 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 Section 5.4.

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 $c' \succeq c$.

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 scope registry.

R5. If $A_r \in c.\mathcal{S}$, update backup scope registry.

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 $m' \succeq m$.

M2. If $A_r = m.S$, update the status of scope in the scope registry.

M3. If $A_r \in m.\mathcal{S}$, update the backup status of the scope in backup scope registry.

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.

5.4 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 section, 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 pending message pool and backup message pool 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 (Chapter 3), 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 pending message pool and backup message pool. 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. See near the end of Subsection 5.5.1 for a concrete example of this scenario.

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.

5.5 Example

We use the same example as in Section 3.3 and Figure 2.2 to illustrate how replicated CPM works.

5.5.1 Replication degree 1

Suppose the replication degree is 1. When OA A_p starts to orchestrate SA p, A_p generates an initial CPM message $c_0 = orch^{A_p}(-)$ that includes an OAG as the left part of Figure 5.2 (without the A_u part, since the replication degree is 1).

When interpreting CPM message $invoke^{A_p}(S_a, a, \bar{a}) \cdot \ldots$, A_p sends an invocation message to SP S_a and stores message $wait^{A_p}(S_a, a) \cdot \ldots$ in its message pool. According to Rule C2, it also sends this CPM message to $c_0.\mathcal{A}$, which is $\{A_b\}$. So now A_b is the backup of A_p for the orchestration of p.

When interpreting message $c = invoke^{A_b}(S_b, b) \cdot \ldots$, A_p sends the CPM message c (corresponding to message 2 in Figure 3.3) to $c.\mathcal{A}^+$, which is $\{A_p, A_b\}$ (Rule C1-b).

When A_b receives the message, it further interprets the message (Rule R.2). A_p stores the message in its pool of backup messages (Rule R.3). Note that when the replication degree is only 1, no remote message is actually sent for the purpose of backup.

The current scope manager $c.S$ is A_p. $c.\mathcal{S}^+ = \{A_p, A_b\}$, $c_0.\mathcal{A} = \{A_b\}$ and $c.\mathcal{A}^+ = \{A_b, A_p\}$. So $c.\mathcal{S}^+ \cup c_0.\mathcal{A} - c.\mathcal{A}^+ = \emptyset$ and not message is sent according

to Rule C1-c.

The table below shows the messages (the head activities of their continuations) and the corresponding backup OAs during the orchestration of p.

Msg	$c_0.head$	$c.head$	$c_0.\mathcal{A}$	$c.\mathcal{A}$
local at A_p	$orch^{A_p}(-)$	$wait^{A_p}(S_a, a)$	A_b	A_b
2	$orch^{A_p}(-)$	$invoke^{A_b}(S_b, b)$	A_b	A_p
local at A_b	$invoke^{A_b}(S_b, b)$	$wait^{A_b}(S_b, b)$	A_p	A_p
3	$invoke^{A_b}(S_b, b)$	$join^{A_p}(-)$	A_p	A_b
4	$orch^{A_p}(-)$	$invoke^{A_c}(S_c, c)$	A_b	A_p
local at A_c	$invoke^{A_c}(S_c, c)$	$wait^{A_c}(S_c, c)$	A_p	A_p
5	$invoke^{A_c}(S_c, c)$	$invoke^{A_d}(S_d, d)$	A_p	A_c
local at A_d	$invoke^{A_d}(S_d, d)$	$wait^{A_d}(S_d, d)$	A_c	A_c
6	$invoke^{A_d}(S_d, d)$	$join^{A_p}$	A_c	A_b
local at A_p	$join^{A_p}(-)$	$eos^{A_p}(-)$	A_b	A_b

To see more specifically how this works, now consider the case where A_d interprets CPM message $c = join^{A_p} \cdot \ldots$, as Figure 5.5 illustrates. A_d sends message c (messages 6 and 6' in Figure 5.5, which correspond to message 6 in Figure 3.3) to $c.\mathcal{A} = \{A_p, A_b\}$ (Rule C1-b).

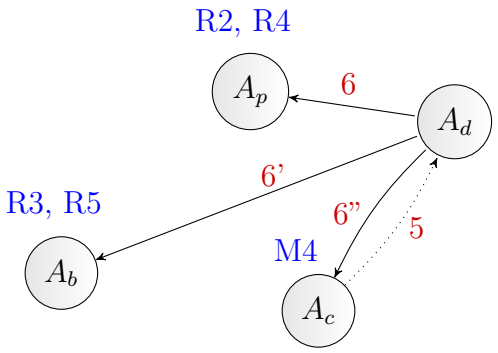


Figure 5.5: Messages from A_d for replicated CPM

The current scope manager $c.S$ is A_p. $c.\mathcal{S}^+ = \{A_p, A_b\}$, $c_0.\mathcal{A} = \{A_c\}$ and $c.\mathcal{A}^+ = \{A_p, A_b\}$. So $c.\mathcal{S}^+ \cup c_0.\mathcal{A} - c.\mathcal{A}^+ = \{A_c\}$ and A_d notifies A_c that the current branch moves to A_p (message 6" in Figure 5.5).

When A_p receives message 6, it further interprets the message (Rule R2) and update the status of the current scope (Rule R4). When A_b receives message 6', it updates the backup messages (Rule R3) and the backup scope status (Rule R5). When A_c receives message 6", it purges the backup messages about p (Rule M4).

Now consider the situation where A_b becomes unavailable before the loop of p finishes. When A_c observes, or is notified of, the unavailability of A_b, it does nothing, except notifying other OA about this (as part of OA network management).

When A_p is notified that A_b becomes unavailable, it finds message $c = wait^{A_b}(S_b, b) \cdot \ldots$ in its backup message pool. It takes over the job of A_b to further orchestrate p (Rule L1). In addition, it adds A_c to $c.\mathcal{A}$ according to the OAG of p and informs $c.\mathcal{A}^+ = \{A_p, A_c\}$ about the latest update of p (Rule L2).

Now consider a particular situation: OA A_d becomes unavailable just after it sends the messages 6, 6' and 6", and the notification of the unavailability arrives at A_c before message 6". In this situation, A_c may take over and repeat the work that A_d had just finished before it became unavailable. A_c repeats the service invocation at S_d. A business-critical service will return with an exception due to the at-most-once semantics. With a proper fault handler, A_c will be able to continue with the execution. The repeated work will eventually arrive at A_p. By checking the timestamp of the message (Rule R1), A_p can figure out that the orchestration of this branch has already passed over this stage. The same is also detected by the scope manager (Rule M1), which is A_p again in this example.

5.5.2 Replication degree 2

Now suppose the replication degree is 2.

When interpreting CPM message $invoke^{A_p}(S_a, a, \bar{a}) \cdot \ldots$, A_p sends an invocation message to SP S_a and stores message $wait^{A_p}(S_a, a) \cdot \ldots$ in its message pool. According to Rule C1, it also sends this CPM message to $c_0.\mathcal{A}$, which is $\{A_b, A_c\}$. So now A_p has two backups A_b and A_c for the orchestration of p.

When interpreting message $c = invoke^{A_b}(S_b, b) \cdot \ldots$, A_p sends the CPM message c (corresponding to message 2 in Figure 3.3) to $c.\mathcal{A}^+$, which is $\{A_p, A_b, A_c\}$ (Rule C1-b).

When A_b receives the message, it further interprets the message (Rule R.2). When A_p and A_c receive the message, they store the message in their pools of backup messages (Rule R.3).

The current scope manager $c.S$ is A_p. $c.\mathcal{S}^+ = \{A_p, A_b, A_c\}$, $c_0.\mathcal{A} = \{A_b, A_c\}$ and $c.\mathcal{A}^+ = \{A_b, A_p, A.c\}$. So $c.\mathcal{S}^+ \cup c_0.\mathcal{A} - c.\mathcal{A}^+ = \emptyset$ and no message is sent according to Rule C1-c.

The table below shows the messages, their head activities and the corresponding backup OAs during the orchestration of p.

Msg	c_0	c	$c_0.\mathcal{A}$	$c.\mathcal{A}$
local at A_p	$orch^{A_p}(-)$	$wait^{A_p}(S_a, a)$	A_b, A_c	A_b, A_c
2	$orch^{A_p}(-)$	$invoke^{A_b}(S_b, b)$	A_b, A_c	A_p, A_c
local at A_b	$invoke^{A_b}(S_b, b)$	$wait^{A_b}(S_b, b)$	A_p, A_c	A_p, A_c
3	$invoke^{A_b}(S_b, b)$	$join^{A_p}(-)$	A_p, A_c	A_b, A_c
4	$orch^{A_p}(-)$	$invoke^{A_c}(S_c, c)$	A_b, A_c	A_p, A_d
local at A_c	$invoke^{A_c}(S_c, c)$	$wait^{A_c}(S_c, c)$	A_p, A_d	A_p, A_d
5	$invoke^{A_c}(S_c, c)$	$invoke^{A_d}(S_d, d)$	A_p, A_d	A_c, A_p
local at A_d	$invoke^{A_d}(S_d, d)$	$wait^{A_d}(S_d, d)$	A_c, A_p	A_c, A_p
6	$invoke^{A_d}(S_d, d)$	$join^{A_p}(-)$	A_c, A_p	A_b, A_c
local at A_p	$eos^{A_p}(-)$	$join^{A_p}(-)$	A_c, A_d	A_c, A_d

Now suppose again that A_b becomes unavailable. When A_c is notified of this, it does nothing. Although A_c finds message $c = wait^{A_b}(S_b, b) \cdot \ldots$ as a backup message, A_c is not the highest ranked available OA in $c.\mathcal{A}^+ - \{A_b\} = \{A_p, A_c\}$ (Rule U1).

When A_p is notified that A_b becomes unavailable, it finds message $c = wait^{A_b}(S_b, b) \cdot \ldots$ in its backup message pool. It takes over the job of A_b to further orchestrate p (Rule L1). In addition, it adds A_d to $c.\mathcal{A}$ according to the OAG of p and informs $c.\mathcal{A}^+ = \{A_p, A_c, A_d\}$ about the latest update of p (Rule L2).

When A_c receives message c from A_p, it does nothing, because it already has c as a backup message (received earlier from A_b) and $c \succeq c$ (Rule R1).

5.6 Related work

The focus of research on reliable services orchestration has been on dealing with failures of constituent services, mostly based on compensation-based recovery [15, 23, 53, 85]. For centralized services orchestration, traditional replication methods can be applied to realize reliable servers. For decentralized service orchestration, little work is done on dealing with failures of orchestration engines or agents.

OSIRIS [68] and OSIRIS-SR [71] take an instantiation-based approach to decentralized services orchestration. The execution nodes in OSIRIS-SR, which correspond to SPs and OAs in our model, are organized in a ring. During the orchestration of a process, the execution migrates from nodes to nodes. One main task of this migration is the transfer of execution data, called whiteboards, from a current node to the succeeding nodes. To avoid losing data due to node crashes, the whiteboards are replicated. The node identifiers in the ring are used to determine on which nodes a whiteboard is replicated. This approach does not

explore the process structures for efficient replication and process execution, as we do.

Several replication schemes have been proposed in the research area of data streams and continuous queries and bear some similarities to our approach. [29] applies 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 or backups is based on peer ids and neighbor proximity of the peer-to-peer network. In peerCQ, selecting replicated peers from the neighbor list localizes the replication process; in addition, peers belongs to this list do not require to remain close to each other geographically, thus the probability of collective failures becomes low. Our replication approach also provides similar advantages by selecting OAs, which are closely associated in a composition. [50] supports an overlay network of peers through which data flow from sensors to data processing programs. Peers are grouped into cells. Active replication is applied to the peers in the same cells to enhance the availability of the data flows.

Active replication ensures a fast reaction to failures, whereas passive reaction usually has a slower reaction to failures. On the other hand, active replication utilizes more resources than passive replication during normal processing. [49] proposes 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. [86] introduces 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.

In [13], authors present an integrated approach for scale out and failure recovery through explicit state management of stateful operators in stream processing systems (SPS). In their implementation, check point states are backed up in upstream operators. Checkpointing is a technique for preserving critical state information. This sort of upstream backup [39] requires nodes to maintain backups until they have been processed by downstream operators or agents. On failure, lost operations are supposed to be replayed by upstream nodes. In general, this suffers from long recovery times when a large set of operations have to be reoperated to restore stateful operators and cannot support state that depends on the complete history. In our approach, scope manager inherently maintains information similar to checkpoints when control flows forwards from one OA to another OA. We maintain and do the selection of backups by a combination of both upstream and downstream orchestration agents. This type of selection procedure reduces overhead significantly and becomes very lucrative, in particular when the replication degree is 1.

Checkpointing is also a method for recovering task execution in case of failure

in multicore processor system. In [31], authors propose a checkpointing based task scheduling algorithm for multicore processor systems. The main idea is: create a checkpoint, and prepare a recovery plan from failure. Checkpoints are created when execution and transfer of the resulting data of each task node to the subsequent processing node is completed. In case of failures (assumed single node failure), it finds the closest ancestor node which is not affected by the failure. Furthermore, recovery plan starts and recovers the processing results from the saved state in that node. During normal execution it has overhead as the communication time between nodes. Similarly, we do recoveries by taking leverage from the scope manager in our context. In addition, we tolerate multiple failures of orchestration agents by also selecting agents from the downstream of any composition.

5.7 Summary

Replicated CPM enhances the availability of OAs when OAs intentionally or accidentally leave the OA network. With replicated CPM, every orchestration activity has a *replication degree* k, meaning that, orchestration activity is assigned with a list of $k + 1$ OAs and can tolerate up to k simultaneous OA crashes. We deigned our selection of backup OAs such that they reuse already stored state information and keep the run-time overhead of services orchestration as low as possible, particularly when the replication degree is 1, there are almost no additional remote messages than a non-replicated orchestration.

Another important property of the backup selection is that the backups of an OA can be unambiguously calculated by any OA at any time of the orchestration. When an OA receives a CPM message, it can handle this message independently regardless of whether the OA is an active OA, a backup OA, an active scope manager or a backup scope manager. We have explicit rules for comparing the causality of messages and purging messages when its necessary.

This chapter also includes examples and shows how we calculate selection of backup OAs, handles CPM messages, while we have replication degree 1 and 2.

Finally, this chapter surveyed related work, focusing on the approaches in distributed orchestration and replication mechanisms for high reliability and availability. The replication in the related work is designed for application domains like continuous queries and data stream processing. There, 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 communication among them.

Chapter 6

Performance Evaluation

This chapter presents the results of some performance studies of our work.

We developed a prototype of OA (as shown in Figure 5.1) in C++. The prototype runs as OMNet++ simulations [77] and messages as described in Chapter 4 are represented in Extensible Markup Language (XML). We choose simulation over running prototype applications for performance study, because with simulation we have better control over a wider variety of configurations and run-time parameters, and thus can obtain better understanding of the factors that influence the performance. Of course, running simulations is more limited and less realistic than running real applications. On the other hand, developing applications is more time consuming and we would have to spend considerable amount of time on implementing the parts that are not relevant to the key research issues of this work.

6.1 Performance of different services orchestration approaches

We first study the performance of different service orchestration approaches, with focus on the scalability to increasing number of services. We compare three orchestration approaches:

- centralized orchestration with central engines,
- decentralized orchestration with continuation-passing messaging, and
- decentralized orchestration where an SA is instantiated and control is pre-allocated to OAs prior to the execution of the SA.

In the table and figures that follow, ctr stands for the centralized approach, cpm for continuation-passing messaging, and dectr for decentralized approach with process instantiation.

In the simulations, service sites are connected with 10mbps links. In centralized and traditional decentralized orchestration, the message size depends on the sizes of the input and output of the service operations. In CPM orchestration, the message size is also dependent on the sizes of the continuations and environments, which change during the execution of the SAs. In the simulations, the orchestration messages are on average 10K bytes with ctr and dectr. The size of a CPM message is on average 10K bytes multiplying the number of invocations to service operations.

An execution of a service program at a service site takes on average 100ms. A central engine takes on average 5ms to dispatch an activity. With cpm and dectr, a service site works both as an SP and an OA. That is, the computational resources of a site are used both for service execution and for services orchestration. A service site takes on average 10ms to interpret a continuation-passing message and 4ms to interpret an invocation message for an instantiated SA. With central engines, a service site takes on average 2ms to dispatch an invocation.

Every SA consists of 4 parallel branches, each consisting of a sequence of 4 service invocations. The service sites are chosen randomly. The second service in the second branch has a 20% chance of throwing a fault.

	successful execution	failed execution
ctr	0.88	0.87
cpm	0.82	1.52
dectr	0.98	1.68

Table 6.1: Response time of processes (in seconds)

Table 6.1 shows response time of the SAs when there is no resource contention. For successful executions, cpm has the shortest response time. For failed executions, ctr outperforms decentralized approaches. The reason is that, invocations of all services are routed via the central engine, so that a fault is notified to any non-faulty branch by the next invocation. With decentralized approaches, a non-faulty branch simply moves forward until a notification message from the scope manager arrives.

To study performance under different workload, we adopt a closed queuing model that keeps a constant system load during each simulation run. The workload of a site is given by its *multiprogramming level*, or MPL, which is the number of concurrent service executions at that site. So MPL 6 means that each site concurrently executes 6 services most of the time. Initially, a fixed number of SAs are

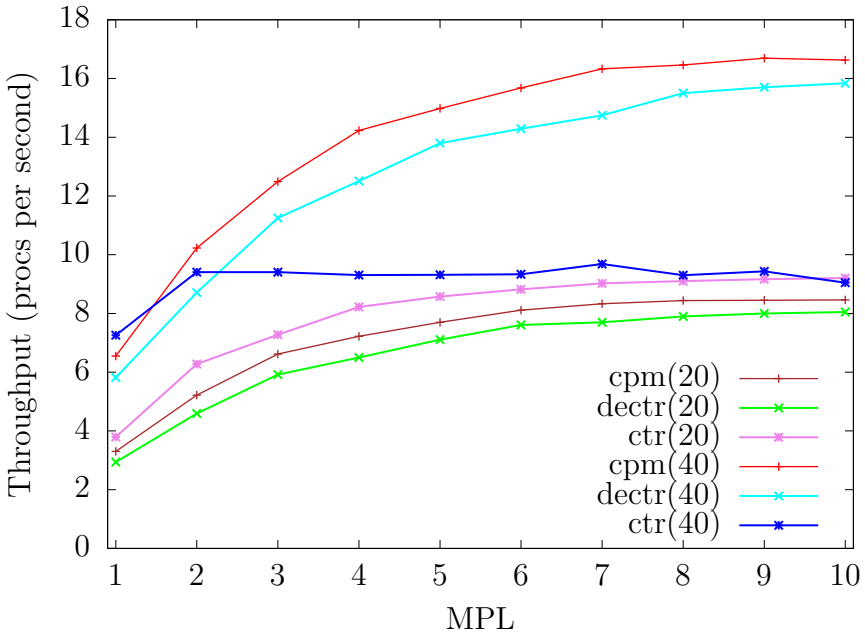


Figure 6.1: Aggregate throughput of all servers

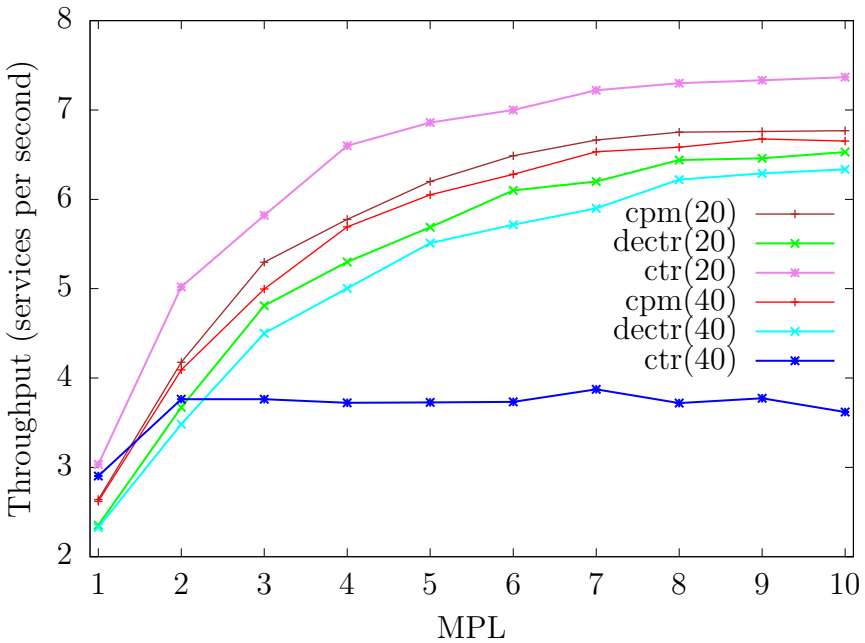


Figure 6.2: Throughput of a service site

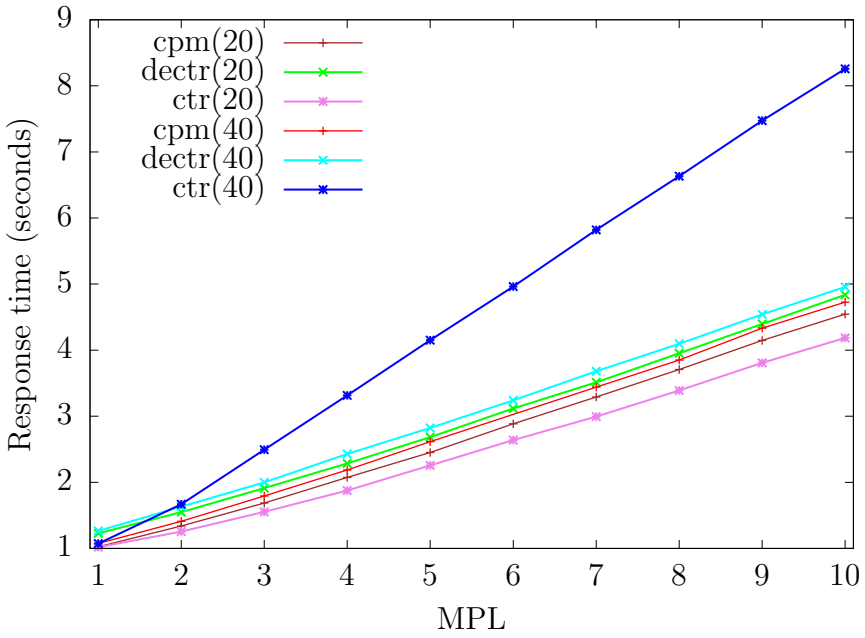


Figure 6.3: SA response time

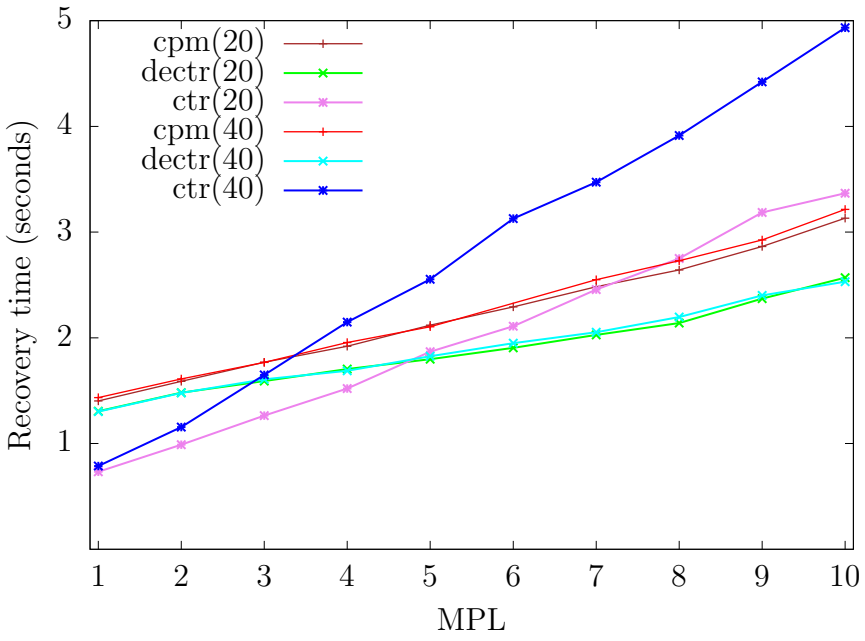


Figure 6.4: SA recovery time

fed into the system. A new SA is started as soon as an existing one terminates.

First, we run simulations with 20 and 40 service sites. In the figures 6.1–6.8, the numbers in parentheses are the numbers of service sites. Figure 6.1 shows the aggregate throughput of all service sites (measured in the number of successfully completed SAs per second). Figure 6.2 shows the throughput of individual service sites (measured in the number of services of successfully completed SAs per second). Figure 6.3 shows the response time of the succeeded SAs.

When there are 20 service sites, ctr outperforms the decentralized ones, with both higher throughput and shorter response time. Note the slight unfairness in this comparison: ctr has dedicated extra resource solely for the purpose of orchestration, i.e., 21 vs. 20 machines.

When there are 40 service sites, the central engine gets congested, whereas with decentralized approaches, throughput still grows with the increase of system load. For decentralized approaches, cpm outperforms dectr, meaning that the overhead due to process instantiation in dectr outweighs the overhead due to larger messages and longer message interpretation time in cpm.

Figure 6.4 shows the recovery time of failed executions. Here, the recovery time is the time from a fault is thrown till the recovery completes, i.e., it includes the time for both fault propagation and compensation of completed services. With the increase of workload, the recovery time of ctr increases faster than the decentralized approaches, indicating that the performance of recovery is more sensitive to the load at the central engine. dectr uses less time for recovery than cpm, because the control for recovery was already allocated during SA instantiation.

To further study the scalability of the different approaches, we next run simulations where the central engine is implemented as a pool of 10 engines. Dispatching the messages to the right engine incurs an extra overhead of 1ms. The number of service sites now increases to 200 and 400.

The performance is shown in Figures 6.5–6.8. As we can see, even when the number of service sites is 200, ctr cannot scale further up after MPL 5. Decentralized approaches scales up much better than centralized ones.

Although in theory it is always possible to scale up a central server with a larger engine pool, it is impractical when the number of active service sites and their workload are dynamic and unknown. With decentralized approaches, response time and throughput of service sites show identical curves with 200 and 400 service sites, meaning that the performance of individual service sites is nearly independent of the number of other service sites around in the world.

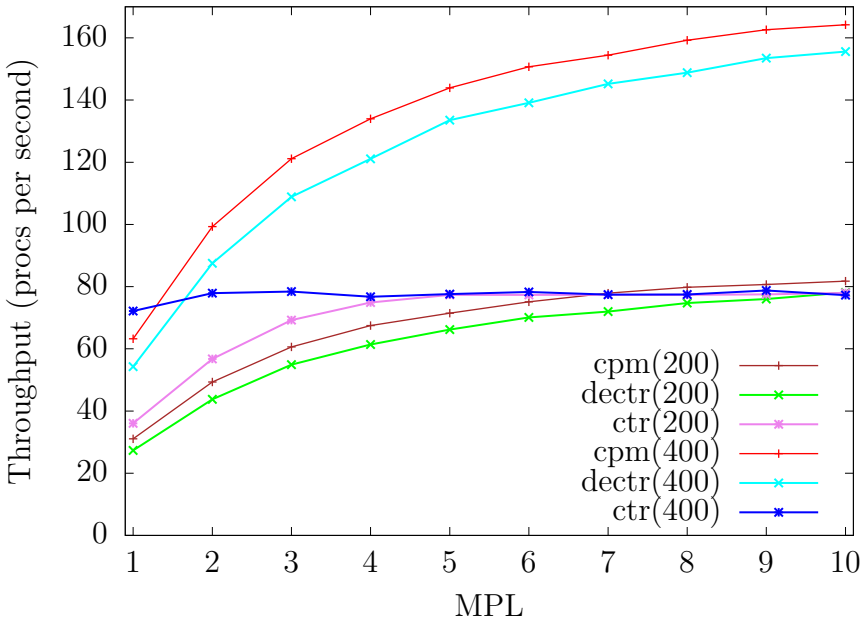


Figure 6.5: Aggregate throughput (pooled ctr)

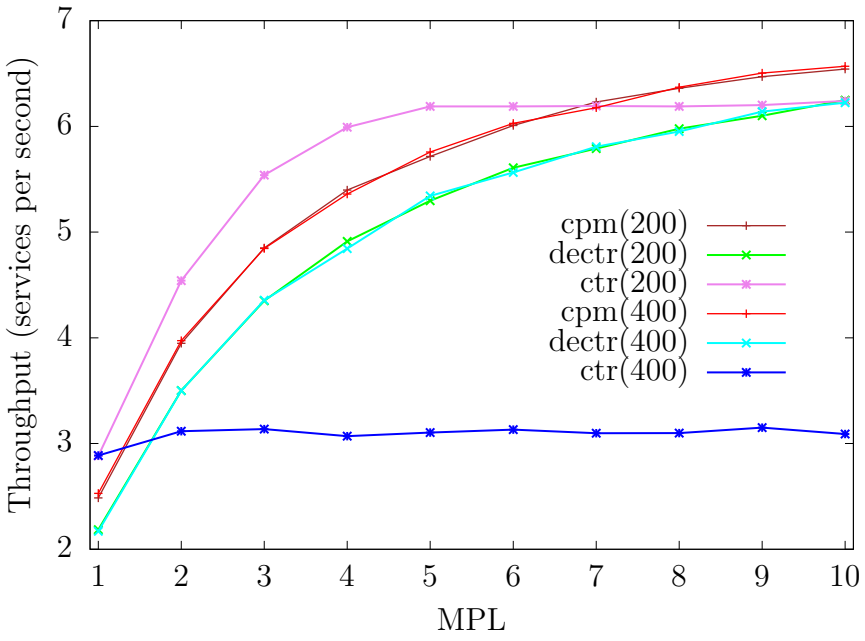


Figure 6.6: Throughput of a service site (pooled ctr)

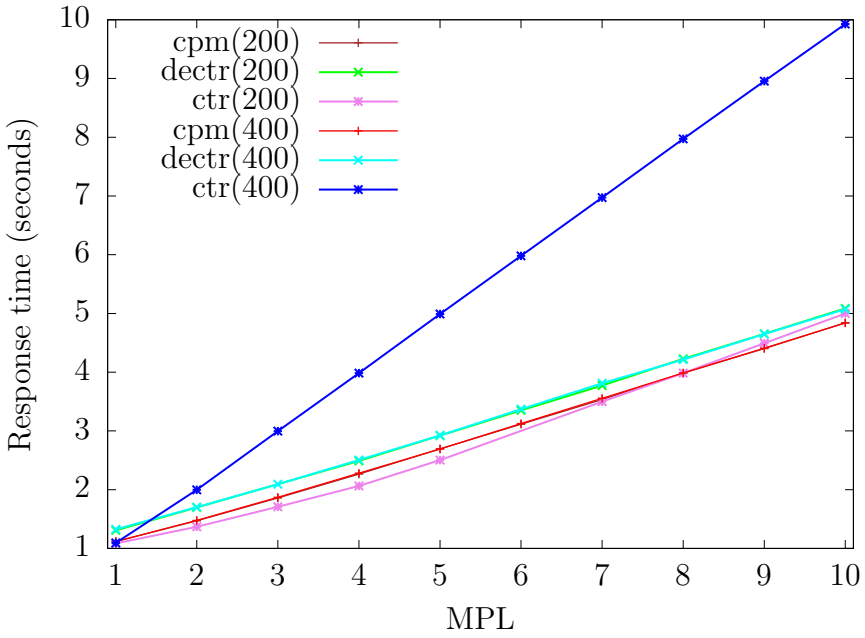


Figure 6.7: SA response time (pooled ctr)

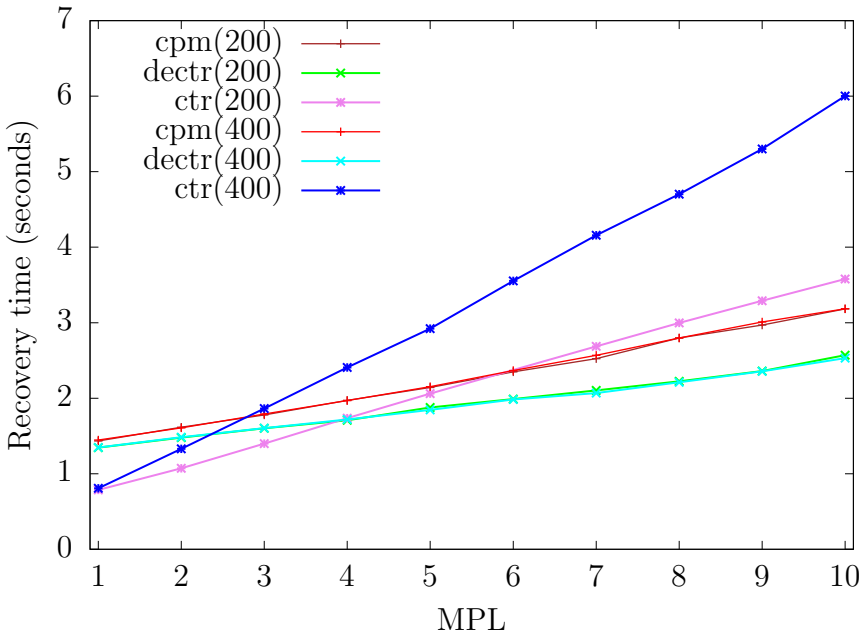


Figure 6.8: SA recovery time (pooled ctr)

6.2 Performance of web mashups

Execution of web mashups can be regarded as a hybrid of centralized and decentralized service orchestration. There is no central orchestration engine, but the hosts of SAs (i.e. mashups) conduct the execution of the SAs in a centralized manner.

Because there is no central engine, there is no single system-wide performance bottleneck. The performance is therefore dependent on the individual hosts of the SAs. However, invoking a remote service is more costly than invoking a local procedure. The response time of an SA may be dependent on the distances of the invoked services.

This section compares the response time of CPM orchestration with executions of web mashups, i.e. orchestrations with SA hosts.

We use the example SA in Figure 2.2 for this study.

The SPs in the SA are chosen randomly from 100 SPs. With CPM orchestration, S_b and A_b are located in the same LAN. We vary the loop size with 1, 10, 100 and 1000 iterations. Figure 6.9 shows the average response time of the successful executions. The performance gain of CPM orchestration increases with the size of the loop. Therefore, if S_b, or the cloud hosting S_b, offers the function of an OA, the overall performance of S_b may appear to be increased significantly.

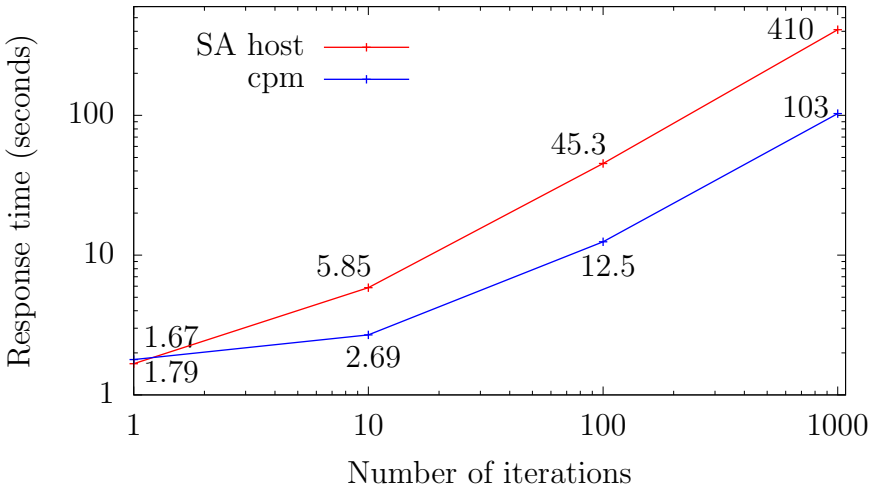


Figure 6.9: Response time of the example SA

Next, we study the response time in different specific scenarios:

1. the SA is orchestrated by the SA host that is near S_b;
2. the SA is orchestrated by the SA host that is far from S_b;

3. the SA is orchestrated with CPM, S_b is far from the SA host and the coverage of S_b is known;

4. the SA is orchestrated with CPM, S_b is far from the SA host, the coverage of S_b is unknown and the algorithm *learnInOrch* is used in the first iterations to learn about S_b (Section 3.5 and Figure 3.5).

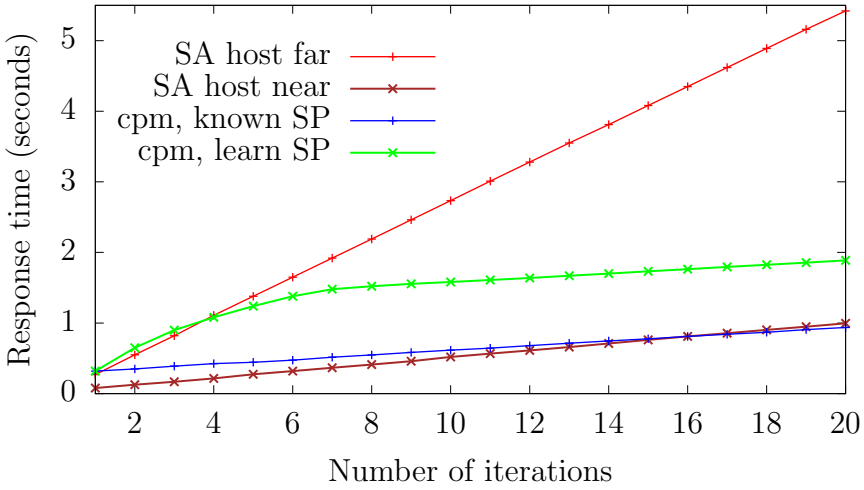


Figure 6.10: Response time of a simple loop

Figure 6.10 shows the result of the experiment where the SA consists of a single loop. When the SA host is near S_b, orchestration by the SA host has short response time (Scenario 1). However, when the SA host is far from S_b and the SA is still orchestrated by the SA host (Scenario 2), the response time grows quickly with the number of iterations of the loop.

When the SA is orchestrated with CPM and the coverage of S_b is known (Scenario 3), the response time of the SA is very close to Scenario 2, although the SA host is far from S_b.

When the coverage of S_b is unknown (Scenario 4), the first few iterations incur longer delays, because the OAs have to do some extra work to learn about the S_b. As soon as the OAs have learned the coverage of S_b, the delays of the following iterations are the same as when the coverage is known. In this particular case, the cost of the learning, i.e. the extra delay in the first few iterations (compared to Scenario 3), is less than 1 second.

6.3 Performance of replicated CPM

Now, 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 SA consists of 4 sequential invocations to service operations at different SPs. These SPs are chosen randomly.

Figure 6.11 shows the aggregate throughput of all SPs (measured in the number of completed SA executions per second). Figure 6.12 shows the average response time of the SAs.

It is not surprising that the higher the replication degree, the more run-time overhead the orchestration has, and thus the lower throughput and longer response time.

Next, we analyze where the run-time overhead comes from.

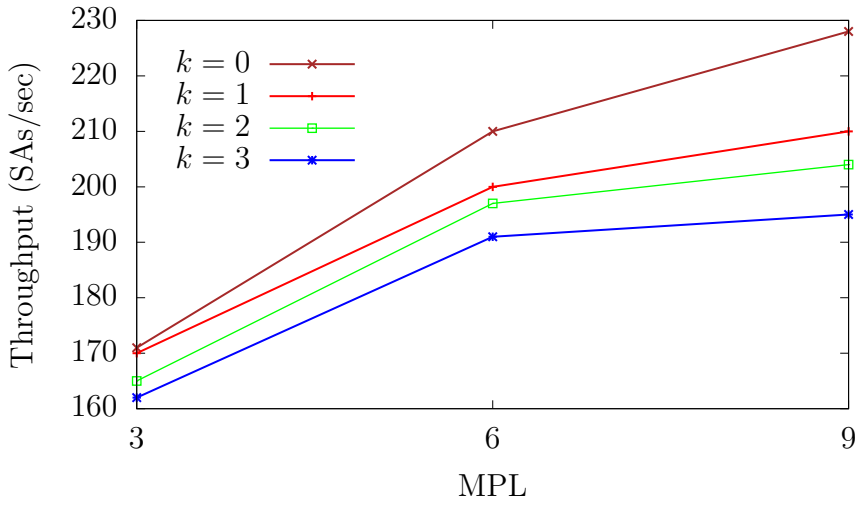


Figure 6.11: Throughput of 100 SPs

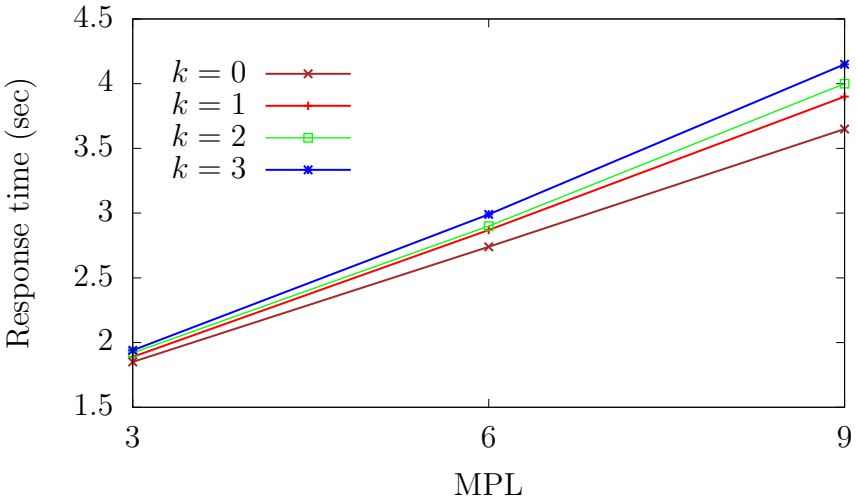


Figure 6.12: Response time of SAs

Figure 6.13 shows the average resource utilization at OAs when the SP MPL is 6. We only show the resource utilization at one particular 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.

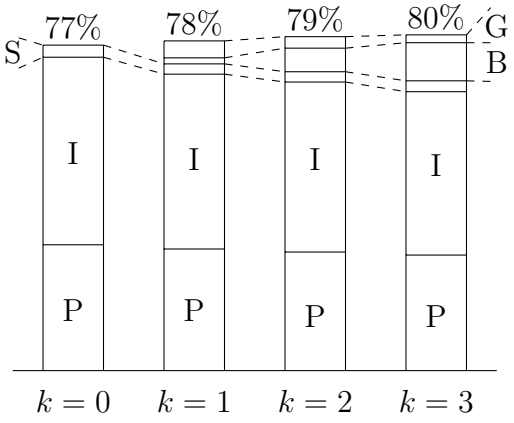
The Figure shows the total resource utilization, like 77% etc., and the portions for specific tasks, like "P" for processing service operations, "I" for message interpretation, and so on.

In Figure 6.13, we can see that as the degree of replication increases, the overall 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 Figures 6.11 and 6.12.

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 Section 5.3. 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 stores 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 Section 5.3 leads to the decrease of "G".

We also experimented with the situation where an OA becomes unavailable. Although we made some observations, we were not able to to draw definitive



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

Figure 6.13: Resource utilization at OAs at MPL 6

conclusions. 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 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 Figure 6.13 that contributed to the increase of load at the remaining OAs.

6.4 Summary

We evaluated the performance of decentralized service orchestration with CPM by running simulations.

First, we compared the scalability of one centralized and two decentralized orchestration approaches. Decentralized approaches are clearly more scalable than centralized ones. This also includes the cases where the central engine is implemented as a pool of servers.

Second, we compared CPM orchestration with web mashups, which can be regarded as a hybrid of centralized and decentralized approaches. Since CPM takes into account the structure of a composition and the geographical locations of the OAs and SPs, when a composition involves repeated invocations to a particular service, CPM orchestration outperforms web mashups.

Finally, we experimented with replicated CPM. Clearly, replication incurs a performance penalty. We further analyzed specifically where replication introduced run-time overheads.

Chapter 7

Conclusion

In this final chapter, we summarize the main contributions of this thesis work, point out its limitations and shed some light on possible future work.

7.1 Contributions

We present the contributions of this thesis work according to the Problem Statement (Section 1.4) in the Introduction of this dissertation.

> **Q1.** *Is it possible to orchestrate open services without a central engine and without pre-allocation of control and resources?*

We proposed continuation-passing messaging (CPM), as a new approach to decentralized services orchestration. Instead of central engines, a set of orchestration agents, which are outside of any administration boundaries, jointly orchestrate open services by exchanging and interpreting orchestration messages. Information that is necessary for the orchestration of composition of open services, including execution plans and run-time states, is encapsulated in CPM messages. Our approach is different from most of the earlier decentralization approaches in that there is no need to instantiate the composite services and pre-allocate resources before their executions. We argue that this is more appropriate for the orchestration of open services.

For the aspects we have investigated, in particular, for the control of executions according the composition specification, our answer to the question is yes, it is possible to orchestrate open services without central engines and without pre-allocation of control and resources.

We also conducted performance studies and compared our approach with centralized approaches and with decentralized approaches that pre-allocate control

and resource before the executions of composite services. The studies show that decentralized approaches are clearly more scalable to the increasing number of concurrent services than the centralized ones.

Q2. *Is it possible to handle exceptions at run time when the execution is dynamically spread around in the distributed environment?*

With service-oriented computing, run-time exceptions are typically captured and handled by central engines or dedicated resources pre-allocated prior to the execution of composite services. We introduced two mechanisms to handle run-time exceptions. First, we dynamically generate recovery plans during the orchestration of composite services and encapsulate them in CPM messages in the form of compensation continuations. Second, we use scope managers to propagate and handle exceptions. When an exception occurs, it is captured locally by the current orchestration agent. The agent can already handle the local part of the exception. In addition, it notifies the enclosing scope manager to further propagate and handle the exception.

So our answer to this question is yes, it is possible to handle exceptions at run time when the execution is dynamically spread around in the distributed environment.

Q3. *Is it possible to tolerate unexpected failures when the execution is dynamic and distributed?*

To deal with the unexpected unavailability of the orchestration agents at run time, we presented a replication scheme that utilizes and expands the run-time status information already spread in the distributed orchestration environment. With a replication degree k, the approach can tolerate up to k simultaneous crashes of orchestration agents. Our replication scheme is special in that it utilizes the decentralized nature of CPM where information about the control and run-time states of an orchestration is already spread and partially replicated. We have also run experiments to study the extra run-time overhead of the replication scheme.

Our answer to this question is yes, it is possible to tolerate unexpected failures when the execution is dynamic and distributed.

Q. *Is it possible to perform reliable orchestration of composite open services?*

We have only investigated some aspects of decentralized orchestration of open services, namely conduct of the execution of a composition according to the specification, handling of exceptions and unexpected failures. We believe these are key to the success for reliable orchestration of composite open services. Our research indicates promising possibilities. Still, our work is very limited and there are a

number of open issues, as discussed in the following sub-sections.

7.2 Limitations

We chose to use simulation to evaluate our work, which is clearly more restricted in some respects than using a working application. We believe this is the right choice, because we can then focus on the core technical issues within a limited time frame.

There are still a number of issues to be addressed before our 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, CPM orchestration can be used as a smart pool of orchestration engines where the orchestration activities are dispatched to the most appropriate engines.

We have currently focused on control flows of composite services. Management of data flows is also important, in particular when the volume of data is significant, such as in data-intensive applications. In our current work, we can use dependency links between activities (Section 4.8) as a way of explicitly defining data flows. More work is needed to reduced the amount of data transfer during service executions.

Currently, the only operation on data is assignment, as is the case in web service standards like WS-BPEL [57]. To develop real-world applications, a richer data-manipulation language is needed.

7.3 Future work

Concurrent updates of data at different parallel branches may leave the data values non-deterministic. We have earlier proposed an approach to data consistence through scope management [84]. We are currently investigating conflict-free data types [69] as a means to achieving data consistence.

Orchestrating services does consume a significant amount of resource. An incentive model that rewards these orchestration-offering service providers would encourage more to offer as orchestration agents. For example, applications that consist of services from these providers should have higher priority when scheduled in orchestration agents and should be more entitled to higher degree of replication.

Our performance study shows that replicated executions incur a performance penalty. Unnecessarily high degree of replication should therefore be avoided. Not all activities in an application have the same requirement on availability. We are interested in an adaptive replication scheme where different activities of the same execution may have different degrees of replication.

More broadly, we have shown that continuations-passing messaging can be regarded as a new form of decentralized execution of programs. It would be interesting to see in which new areas this new form of program execution can find its application.

Appendix: Publications

This appendix presents an overview of the publications which this dissertation is based on.

Chapters 3 and 4 are based on Papers I, II and III.
Chapter 5 is based on Paper IV and V.
Chapter 6 is based on the performance results in Papers I, II and V.

Paper I

I. Weihai Yu and Abul Ahsan Md Mahmudul Haque. Decentralised web-services orchestration with continuation-passing messaging. *International Journal of Web and Grid Services* (IJWGS), 7[3], pages 304–330, 2011.

In this paper, we presented a new approach to decentralized process orchestration using continuation-passing messaging. This is an early version of our work, where the service providers are themselves responsible for the orchestration of the processes they involve.

Service-oriented architecture is primarily adopted for cost-effective construction of enterprise applications. Meanwhile, many web applications start to provide open APIs to be readily applicable by a wider range of applications. Whilst the traditional centralized approaches to orchestrating composite services are successful in enterprise service-oriented architectures, they are subject to serious limitations for orchestrating services in the wider range of open web applications. Dealing with these limitations calls for decentralized services orchestration. However, existing decentralized approaches are themselves faced with a number of technical challenges, due primarily to lack of overview of dynamic process status. Tasks like fault handling and recovery are generally considered difficult for decentralized approaches. We introduced a decentralized approach based on continuation-passing messaging where control and status information about service executions is carried in messages for services orchestration. We

demonstrated that this new decentralized approach is capable of handling dynamic run-time tasks like fault handling and recovery. Our experimental results show performance advantage of the approach, both in normal executions and in case of service failures.

Papers II and III

II. Abul Ahsan Md Mahmudul Haque and Weihai Yu. Peer-to-peer orchestration of web mashups. *International Journal of Adaptive, Resilient and Autonomic Systems* (IJARAS), 5[3], pages 40–60, 2014.

This paper explores the idea of services orchestration, specially web mashups, which takes leverage from a network of agents. This is an extension to our earlier work which looks too intrusive as it requires that service providers support message interpretation. Although this might be arguably acceptable for enterprise applications, it is too strong as a requirement for open services.

Web mashups are web applications built on top of external web services through their open APIs. As mashups are becoming increasingly complex, there is a need for systematic support for their development and orchestration. This paper presents a peer-to-peer approach to mashup orchestration where a network of agents carry out orchestration using continuation-passing messaging. The approach supports exception handling and recovery. Our experimental results show clear performance gains of the approach over traditional centralized orchestration in service-oriented computing and orchestration done by application servers hosting mashups.

This paper was initially published in

III. Abul Ahsan Md Mahmudul Haque, Weihai Yu, Anders Andersen, and Randi Karlsen. Peer-to-peer orchestration of web mashups. In *Proceedings of the 14th International Conference on Information Integration and Web-based Applications & Services*, (iiWAS'12), pages 294–298, ACM, 2012.

Later the paper was among the short list of the papers nominated for the IJARAS Journal.

Papers IV and V

IV. Abul Ahsan Md Mahmudul Haque and Weihai Yu. Towards a dynamic replication scheme for processes with open services. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications* (SOCA'13), pages 366–370, Dec. 2013.

In this paper, we present the idea of a dynamic replication scheme for a decentralized orchestration approach, where a network of agents collectively orchestrate open services using continuation-passing messaging. The scheme utilizes the knowledge about the control structures that are encapsulated in messages and the run-time states that are already spread in the distributed environment to enhance the reliability of the processes.

Later, we extended our work and published the following paper which received "Best Paper Award" in the conference.

V. Abul Ahsan Md Mahmudul Haque and Weihai Yu. Decentralized and reliable orchestration of open services. In *The Sixth International Conferences on Advanced Service Computing SERVICE COMPUTATION*, pages 1–8, May 2014.

This paper enhances our previous work. We developed an OA prototype for replicated CPM and studied the performance of OAs with different degrees of replication and at different workload.

Next, we include Papers I, II and V for further convenience.

References

[1] Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. *submitted to IEEE Expert*, 1997. 54

[2] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. 17

[3] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Trans. Software Eng.*, **24**[2]:149–159, 1998. 18

[4] Anders Andersen. SNOOP: privacy preserving middleware for secure multi-party computations. In *Proceedings of the 13th Workshop on Adaptive and Reflective Middleware, ARM@Middleware 2014, Bordeaux, France, December 8-12, 2014*, pages 8:1–8:6, 2014. 7

[5] Daniel Barbará, Sharad Mehrotra, and Marek Rusinkiewicz. INCAs: Managing dynamic workflows in distributed environments. *Journal of Database Management*, **7**[1]:5–15, 1994. 55

[6] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *2006 IEEE International Conference on Web Services (ICWS'06)*, pages 63–71, Sept 2006. 17

[7] Joel F. Bartlett. A nonstop kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 22–29, New York, NY, USA, 1981. ACM. 17

[8] Boualem Benatallah, Marlon Dumas, and Quan Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, **17**[1]:5–37, 2005. 13, 54, 55

[9] Tim Berners-Lee. The World Wide Web: Past, present and future. https://www.w3.org/People/Berners-Lee/1996/ppf.html, 1996. Accessed: 2017-01-05. 1

[10] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 90–99, New York, NY, USA, 1983. ACM. 17

[11] László Böszörményi, Robert Eisner, and Herbert Groiss. Adding distribution to a workflow management system. In *DEXA Workshop*, pages 17–21, 1999. 13, 55

[12] Sonja Buchegger, Doris Schiöberg, Le-Hung Vu, and Anwitaman Datta. Peerson: P2P social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems(SNS)*, pages 46–52, 2009. 4

[13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM. 70

[14] Girish Chafle, Sunil Chandra, Pankaj Kankar, and Vijay Mann. Handling faults in decentralized orchestration of composite web services. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *ICSOC*, **3826** of *Lecture Notes in Computer Science*, pages 410–423. Springer, 2005. 13, 55

[15] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *Proceedings of the 13th international conference on World Wide Web (WWW 2004)*, pages 134–143, 2004. 14, 69

[16] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, **3**[1]:63–75, 1985. 18

[17] M. Chereque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active replication in delta-4. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 28–37, 1992. 18

[18] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, pages 85–98, 2004. 33

[19] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, **34**[3]:375–408, September 2002. 18

[20] Nicole C. Engard. *Library mashups : exploring new ways to deliver library data*. Information Today, Medford, 2009. 11

[21] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, second edition, 2005. 1

[22] Georgios John Fakas and Bill Karakostas. A peer to peer (P2P) architecture for dynamic workflow management. *Information & Software Technology*, **46**[6]:423–431, 2004. 13, 54, 55

[23] Georgios John Fakas and Bill Karakostas. A peer to peer (P2P) architecture for dynamic workflow management. *Information & SW Technology*, **46**[6]:423–431, 2004. 69

[24] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–217, August 1986. 56

[25] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 407–416, New York, NY, USA, 2000. ACM. 2

[26] Pierre Fraigniaud and Philippe Gauron. D2B: A de bruijn based content-addressable network. *Theor. Comput. Sci.*, **355**[1]:65–79, 2006. 33

[27] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. MIT Press, 2008. 55

[28] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, **31**[1]:1–26, March 1999. 17

[29] B. Gedik and Ling Liu. A scalable peer-to-peer architecture for distributed information monitoring applications. *IEEE Transactions on Computers*, **54**[6]:767–782, 2005. 70

[30] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks: Algorithms and evaluation. *Perform. Eval.*, **63**[3]:241–263, 2006. 33

[31] Shohei Gotoda, Minoru Ito, and Naoki Shibata. Task scheduling algorithm for multicore processor system for minimizing recovery time in case of single node fault. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 260–267, Washington, DC, USA, 2012. IEEE Computer Society. 71

[32] P. Krishna Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany*, pages 381–394, 2003. 33

[33] Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *Software Engineering, IEEE Transactions on*, **26**[10]:943–958, 2000. 16

[34] Grégoire Hamon. A denotational semantics for stateflow. In Wayne Wolf, editor, *EMSOFT*, pages 164–172. ACM, 2005. 56

[35] Abul Ahsan Md Mahmudul Haque and Weihai Yu. Towards a dynamic replication scheme for processes with open services. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 366–370, Dec 2013. 6, 7

[36] Abul Ahsan Md Mahmudul Haque and Weihai Yu. Decentralized and reliable orchestration of open services. In *The Sixth International Conferences on Advanced Service Computing SERVICE COMPUTATION 2014*, pages 1–8, May 2014. 6, 7

[37] Abul Ahsan Md Mahmudul Haque and Weihai Yu. Peer-to-peer orchestration of web mashups. *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, **5**[3]:40–60, July 2014. 6

[38] Abul Ahsan Md Mahmudul Haque, Weihai Yu, Anders Andersen, and Randi Karlsen. Peer-to-peer orchestration of web mashups. In *Proceedings of the 14th International Conference on Information Integration and Web-based Applications & Services*, IIWAS '12, pages 294–298, New York, NY, USA, 2012. ACM. 6

[39] J. H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)*, pages 779–790, April 2005. 70

[40] Kjetil Jacobsen. *Practical Fault-Tolerance for Mobile Agents*. PhD thesis, university of Tromsø, Faculty of Science and Technology, Department of Computing Science, 2011. 6

[41] Suresh Jagannathan. Communication-passing style for coordination languages. In David Garlan and Daniel Le Métayer, editors, *COORDINATION*, **1282** of *Lecture Notes in Computer Science*, pages 131–149. Springer, 1997. 56

[42] Peter John Landin. The mechanical evaluation of expressions. *Computer Journal*, **6**[4]:308–320, 1964. 56

[43] Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, second edition, 2005. 10

[44] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In Gustavo Alonso, editor, *Middleware*, **3790** of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2005. 55

[45] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. Adaptive content-based routing in general overlay topologies. In Valérie Issarny and Richard E. Schantz, editors, *Middleware*, **5346** of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2008. 55

[46] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings.*, 2005. 33

[47] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, **7**[1-4]:72–93, 2005. 33

[48] Dragos-Anton Manolescu. Workflow enactment with continuation and future objects. In *OOPSLA*, pages 40–51. ACM, 2002. 55

[49] 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)*, pages 21–30, 2011. 70

[50] 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)*, pages 313–317, 2010. 70

[51] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, **8**[6]:51–59, 11 2004. Opphavsrett - Copyright IEEE Computer Society Nov 2004; Sist oppdatert - 2010-06-06; CODEN - IESEDJ. 9

[52] Thorsten Möller and Heiko Schuldt. A platform to support decentralized and dynamically distributed P2P composite OWL-S service execution. In Karl M. Göschka, Schahram Dustdar, Frank Leymann, and Vladimir Tosic, editors, *MW4SOC*, ACM International Conference Proceeding Series, pages 24–29. ACM, 2007. 13, 55

[53] Thorsten Möller and Heiko 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)*, pages 24–29, 2007. 69

[54] Luc Moreau. The PCKS-machine: An abstract machine for sound evaluation of parallel functional programs with first-class continuations. In Donald Sannella, editor, *ESOP*, **788** of *Lecture Notes in Computer Science*, pages 424–438. Springer, 1994. 56

[55] Peter Muth, Dirk Wodtke, Jeanine Weissenfels, Angelika Kotz Dittrich, and Gerhard Weikum. From centralized workflow specification to distributed workflow execution. *J. Intell. Inf. Syst.*, **10**[2]:159–184, 1998. 13, 54

[56] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing execution of composite web services. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 170–187. ACM, 2004. 13, 54

[57] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, April 2007. 10, 15, 35, 36, 38, 87

[58] Cesare Pautasso, Thomas Heinis, and Gustavo Alonso. Autonomic resource provisioning for software business processes. *Information & Software Technology*, **49**[1]:65–80, 2007. 54

[59] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, **36**[10]:46–52, 2003. 9

[60] Heorhi Raik. *Service Composition in Dynamic Environments: From Theory to Practice*. PhD thesis, University of Trento, 2012. 10

[61] Brian Randell. System structure for software fault tolerance. *IEEE Trans. Software Eng.*, **1**[2]:221–232, 1975. 17, 18

[62] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001. 33

[63] Hajo A. Reijers. *Design and Control of Workflow Processes: Business Process Management for the Service Industry*. Springer-Verlag, Berlin, Heidelberg, 2003. 10

[64] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*, pages 329–350, 2001. 33

[65] Bart Willem Schermer et al. *Software agents, surveillance, and the right to privacy: a legislative framework for agent-enabled surveillance*. Leiden University Press, 2007. 21

[66] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, **22**[4]:299–319, December 1990. 18

[67] Joerg Schneider, Barry Linnert, and Lars-Olof Burchard. Distributed workflow management for large-scale grid environments. In *SAINT*, pages 229–235. IEEE Computer Society, 2006. 13, 55

[68] Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans-Jörg Schek. Scalable peer-to-peer process management - the OSIRIS approach. In *ICWS*, pages 26–34. IEEE Computer Society, 2004. 13, 33, 54, 69

[69] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, **6976** of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. 87

[70] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001. 33

[71] Nenad Stojnic and Heiko Schuldt. OSIRIS-SR: a scalable yet reliable distributed workflow execution engine. In *Proceedings of the 2nd ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, SWEET@SIGMOD 2013, New York, New York, USA, June 23, 2013*, pages 3:1–3:12, 2013. 33, 69

[72] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, **3**[3]:204–226, 1985. 18

[73] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. 14

[74] Francesco Tiezzi. *Specification and Analysis of Service-Oriented Applications*. PhD thesis, Universita degli ‘ Studi di Firenze, April, 2009. 1

[75] Eneia Todoran and Nikolaos Papaspyrou. Continuations for parallel logic programming. In *PPDP*, pages 257–267, 2000. 56

[76] Sipat Triukose, Zhihua Wen, and Michael Rabinovich. Measuring a commercial content delivery network. In *Proceedings of the 20th International Conference on World Wide Web*, WWW ’11, pages 467–476, New York, NY, USA, 2011. ACM. 33

[77] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In Sándor Molnár, John R. Heath, Olivier Dalle, and Gabriel A. Wainer, editors, *SimuTools*, page 60. ICST, 2008. 72

[78] The World Wide Web Consortium (W3C). Web Services activity. https://www.w3.org/2002/ws/. Accessed: 2016-04-11. 11

[79] The World Wide Web Consortium (W3C). Web Services glossary. https://www.w3.org/TR/ws-gloss. Accessed: 2017-01-06. 2, 11

[80] Mathias Weske. *Bussiness Process Management: Concepts, Languages, Architectures*. Springer, 2007. 10

[81] M. Wieland, K. Gorlach, 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 '09)*, pages 109–118, 2009. 4

[82] Xinfeng Ye. Towards a reliable distributed web service execution engine. In *ICWS*, pages 595–602. IEEE Computer Society, 2006. 13, 55

[83] Jin Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding mashup development. *Internet Computing, IEEE*, **12**[5]:44 –52, sept.-oct. 2008. 4, 12

[84] Weihai Yu. Decentralized orchestration of bpel processes with execution consistency. In *Advances in Data and Web Management, Joint International Conferences, APWeb/WAIM 2009*, pages 665–670. Springer-Verlag, 2009. 87

[85] Weihai Yu and Abul Ahsan Md Mahmudul Haque. Decentralised web-services orchestration with continuation-passing messaging. *International Journal of Web and Grid Services*, **7**[3]:304–330, 2011. 6, 69

[86] Zhe Zhang, Yu Gu, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. A hybrid approach to high availability in stream processing systems. In *Proceedings of IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 138–148, 2010. 70

[87] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Anthony D. Joseph, and John Kubiatowicz. Exploiting routing redundancy via structured peer-to-peer overlays. In *11th IEEE International Conference on Network Protocols (ICNP 2003), 4-7 November 2003, Atlanta, GA, USA*, pages 246–257, 2003. 33