NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

DIPLOMA THESIS

Decentralized Business Process Execution in Peer-to-Peer Systems

Ioannis E. Pogkas

Supervisor: Aphrodite Tsalgatidou, Associate Professor NKUA

Technical Support: Michael Pantazoglou, PhD Research Associate NKUA

ATHENS
November 2011

DIPLOMA THESIS

Decentralized Business Process Execution in Peer-to-Peer Systems

Ioannis E. Pogkas
R.N.: M948

SUPERVISOR:

Aphrodite Tsalgatidou, Associate Professor NKUA

TECHNICAL SUPPORT:

Michael Pantazoglou, PhD Research Associate NKUA

Abstract

Business Process Execution Language (BPEL) has become a standard for describing the interactions between business processes. Until recently, only centralized BPEL engines were used to orchestrate the process interactions, while scalability and robustness were addressed via engine replication. To address these issues we propose a fully decentralized solution: by employing a content-based publish/subscribe mechanism on top of a distributed hash table network of peers, we specify a distributed orchestration engine. Furthermore, we extend a previously proposed model that maps BPEL activities into a subscription language, thereby decentralizing business process execution. The publish/subscribe mechanism provides efficient and flexible means for information producers and consumers to exchange data, while the underlying peer-to-peer topology offers scalable query and message propagation. An implementation of the proposed approach is provided and tested over the *PeerSim* simulator. We evaluated our system in terms of efficiency and effectiveness, i.e., scalability, robustness, and overhead.

SUBJECT AREA: Peer-to-Peer Systems, Distributed Systems, Business Processes

Keywords: BPEL, Orchestration Engines, Business Process Execution, Publish/Subscribe systems, Peer nodes, PeerSim

Περίληψη

Η Business Process Execution Language (BPEL) είναι μια πρότυπη γλώσσα για την περιγραφή της αλληλεπίδρασης των επιχειρησιακών διαδικασιών. Μέχρι πρόσφατα, για την ενορχήστρωση των αλληλεπιδράσεων μεταξύ των διαδικασιών χρησιμοποιούνταν μόνο κεντρικοποιημένες BPEL μηχανές, ενώ η ανάγκη για μεγαλύτερη κλιμάκωση και ευρωστία αντιμετωπίζονταν με τη χρήση πολλαπλών παρόμοιων μηχανών. Για τη διευθέτηση των παραπάνω απαιτήσεων προτείνουμε μια πλήρως κατανεμημένη λύση: τη χρήση ενός μηχανισμού δημοσιεύσεων/συνδρομών πάνω από ένα δομημένο δίκτυο ομότιμων κόμβων, το οποίο να βασίζεται στη χρήση πινάκων κατακερματισμού, για την κατασκευή μιας κατανεμημένης μηχανής ενορχήστρωσης διαδικασιών.

Επιπλέον, επεκτείνουμε ένα μοντέλο που είχε προταθεί παλαιότερα για την κατανεμημένη εκτέλεση επιχειρησιακών διαδικασιών και τα οποίο αντιστοιχούσε δραστηριότητες της BPEL σε μια γλώσσα δημοσιεύσεων/συνδρομών. Ο μηχανισμός δημοσιεύσεων/συνδρομών προσφέρει στους παραγωγούς και τους καταναλωτές της παραγόμενης πληροφορίας έναν ευέλικτο και αποδοτικό μηχανισμό για την ανταλλαγή δεδομένων, ενώ η τοπολογία ομότιμων κόμβων προσφέρει τη δυνατότητα επερωτήσεων και δρομολόγησης μηνυμάτων ακόμα και σε δίκτυα με υψηλή κλιμάκωση. Παρουσιάζουμε μια υλοποίηση της προτεινόμενης προσέγγισης και την αξιολογούμε με την χρήση του προσομοιωτή PeerSim.

Αξιολογούμε το σύστημα ως προς την αποδοτικότητα και την αποτελεσματικότητα του, εξετάζοντας την κλιμάκωση, την ευρωστία, και το κόστος με βάση το πλήθος των παραγόμενων μηνυμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Συστήματα ομότιμου προς ομότιμου, Κατανεμημένα συστήματα, Επιχειρησιακές διαδικασίες

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: BPEL, Μηχανές ενορχήστρωσης, Εκτέλεση επιχειρησιακών διαδικασιών, Συστήματα δημοσίευσης/συνδρομής, Ομότιμοι κόμβοι, PeerSim

Dedicated

To my parents Evangelo and Mairy
for their love, constant support, and
for bringing out what is best in me.

To Basil, my best friend and beloved cousin
Der Anfang und das Ende ist eine
Erfindung des Menschen.

Acknowledgements

First and foremost, I would like to thank my supervisor, Mrs Afrodite Tsalgatidou, for giving me the opportunity to work with her and for her patience, trust, and kindness. Her advices and comments on this thesis helped me to improve myself and to push this work one step forward.

Second, I would like to thank Mrs Mema Roussopoulos, Assistant Professor of the department of Informatics and Telecommunications, for being the examiner of this thesis and for proving useful comments during the presentation of this work.

Third, I would like to thank Mr Alex Delis, Professor of the department of Informatics and Telecommunications, for being an excellent tutor during my postgraduate studies, by helping me to become better as a person, scientist, and professional.

Then, I would like to thank all my dear colleagues and co-workers in the s^3lab. Michael Pantazoglou for his patience to fully review this thesis and for providing insightful comments and corrections. Pigi Kouki, for having the kindness to help me with her thoroughly review and critical comments, especially in the early stages of this work, and George Athanasopoulos for helping me with useful suggestions and for proposing interesting future directions.

Last, but not least I thank all my co-workers and fellow programmers at the sinastria/syntech, Manolis[2], Spyros, Daniel, Alexandros[2], Dimitris, and Lakis, for guiding me through my first steps as a professional programmer and for all the great time that we had together.

Contents

List of Figures

List of Tables

Preface

This thesis has been written during my postgraduate studies in the Department of Informatics & Telecommunications, in the program of Computer Systems Technology, 2009–2010. The main supervisor of this thesis was Mrs Afrodite Tsalgatidou Associate Professor of the department of Informatics and Telecommunications, while technical support was offered by the postdoctoral research associate of the s^{3}lab[1], Mr Michael Pantazoglou. Mrs Mema Roussopoulos, Assistant Professor of the department of Informatics and Telecommunications, was the examiner of this thesis.

This thesis presents a distributed architecture for executing business processes composed of Web services. We evaluate the design and implementation of a distributed engine based on the Peer-to-Peer (P2P) architecture, that leverages a publish/subscribe (pub/sub) messaging pattern in order to coordinate the service execution. This architecture differentiates our solution from the current ones as it requires no administration and can offer efficient business process deployment and execution based on locality metrics.

During that period, I was also involved in the EU funded project ENVISION[2] (ENVIronmental Services Infrastructure with ONtologies) and my association with that project provided useful insights about the directions which I had to follow in my work.

[1] http://s3lab.di.uoa.gr/

[2] http://www.envision-project.eu/

Chapter 1

Introduction

Service oriented architecture (SOA) is a software architecture for building enterprise applications. SOA implements business processes or services (i.e. logical encapsulations of business functions) using a set of loosely coupled black-box components coordinated to deliver a well-defined level of service.

The SOA approach allows businesses to leverage existing assets and to easily evolve by supporting new operations. This happens in such a way that an agile business can quickly adapt its processes to an ever changing landscape of opportunities, priorities, partners, and competitors. Foremost, SOA provides separation of concerns (i.e. the separation of business logic from computer logic), therefore enabling an organization to make business decisions supported by technology, instead of making business decisions determined by or constrained by technology. Additionally, SOA enables businesses to leverage existing investments by allowing them to reuse existing applications, and promises interoperability between heterogeneous applications and technologies. Finally, SOA requires the use of acceptable industry standards to link the existing software assets together, like Web Services [BHM+04], thus providing a level of flexibility that wasn't possible before.

The role of Web Services into SOA is of most importance as they provide the realization of SOA. Web Services are pieces of software that use standard web interfaces to communicate with other software through Web interfaces. In short, Web services are software systems designed to support interoperable machine to machine interaction over a network. Today, the proliferation of Web Services standards reflects the demand for distributed enterprise applications that communicate with software services provided by vendors and clients.

For example, an online retailer company (e.g. Amazon[1]) may use the services of a partner shipping company (e.g. DHL[2]) such as a service that allows customers to track the delivery status of the ordered products. For this purpose, the shipping company would expose a component that allows its partners to retrieve delivery status informa-

[1] http://www.amazon.com
[2] http://www.dhl.com

tion. Other external services the retailer may use include a payment service (such as PayPal[3]), a service that provides products description, and a component that accepts customers reviews. In addition, the retailer may use services developed internally, such as a user interface engine (to render graphical interfaces for various devices such as smartphones or tablet computers) and an authentication service. As these components are developed to be loosely coupled it becomes easier to design, develop, modify, and maintain the overall application.

Modern globalized industries have business processes that consist of complex interactions among a large set of geographically distributed services. These services are commonly developed and maintained by various organizations. They can be very large, long running, can manipulate vast quantities of data, and may require thousands or millions of process instances. These business processes may be distributed, e.g. department-level processes, utilizing dozen of activities, or there may be global processes composed from the department-level ones while being geographically distributed. The industries may require thousands of instances of these processes to be executed concurrently at any time. In a SOA-based architecture, the coordination and execution of such large processes, involving dozen of loosely-coupled collaborating parties, is the natural fit for a distributed execution engine.

1.1 Problem Statement

The Web Services Business Process Execution Language (WS-BPEL or simply BPEL) [AAA+07] specification defines how Web services can be composed to orchestrate long running business processes. It is an OASIS[4] standard and was originally created by BEA[5], IBM[6], and Microsoft[7]. BPEL enables developers to specify how information flows between Web services during long-lasting business processes by including support for loops, conditional cases, synchronous and asynchronous communication, concurrent activities, error handling, and recovery. A business process description in BPEL is interpreted by a BPEL orchestration engine. The latter is responsible for carrying out the processes execution and maintaining the state associated with the process instances. Typically, a single centralized engine is deployed to manage an application and scalability is addressed by replicating the engine.

Existing BPEL orchestration engines [ODE11, Act11, Ser11, jBo11, IBM11] support clustering in order to optimize and ensure efficient throughput on highly available systems. When a business process needs to be scaled to meet heavier processing needs, the clustering algorithm automatically distributes processing across multiple engines. Nevertheless, the centralized and clustered approaches have several drawbacks, as they:

[3] https://www.paypal.com
[4] http://www.oasis-open.org
[5] http://www.oracle.com/us/corporate/Acquisitions/bea/index.html
[6] http://www.ibm.com
[7] https://www.microsoft.com

- *Exhibit low scalability.* The centralized orchestration engine becomes a scalability bottleneck as it fails to support large-scale business processes.
- *Lack proximity mechanisms.* In case of data intensive processes the network must transfer an excessive amount of data among geographically dispersed endpoints.
- *Provide coarse-grained mapping.* They provide only coarse-grained mapping of processes instances on computational resources.

The distributed orchestration architectures that have recently emerged [LMJ10, NCS04, BMM05] offer a different approach. Their runtime orchestrates business processes by distributing process execution across several light-weight agents. The distributed architecture is congruent with an inherently distributed enterprise where business processes are geographically dispersed and coordinating partners have to communicate across administrative domains. Furthermore, this solution provides several advantages: a) it removes the scalability bottleneck of a centralized orchestration engine; b) it offers additional efficiencies by allowing portions of processes to be executed close to the data they operate on (thereby conserving data and control traffic), and c) it supports flexible mappings onto heterogeneous platforms and resources, permitting the system to shape itself from a centralized to a fully distributed configuration.

Nevertheless, the existing distributed solutions fail to fully solve many aspects of the above problems, as they:

- *Require specialized agents.* They require the use of special-purpose agents that will act as broker nodes or will execute specific process activities. This approach increases the heterogeneity of the available resources and makes their management more challenging.
- *Require network administration.* They require custom network configuration by a network administrator to allow portions of processes to be executed close to the data they operate on. Thus this approach is static, human intensive, and clearly not optimal in terms of time, machine, and human resources utilization.
- *Provide only static mapping.* Most of them provide a static assignment of the BPEL processes portions to the network agents. Thus, they fail to facilitate dynamic mapping based on the network condition and the agents utilization state.

1.2 Proposed Solution

Content-based pub/sub systems are verified to be an ideal solution for distributed workflow management since they provide a loosely coupled messaging infrastructure [LMJ10]. As shown in Figure 1.1, they consist of three major components: publishers, subscribers, and brokers. Publishers send data to the system as publications, while subscribers express their interest in specific information by issuing subscriptions, and

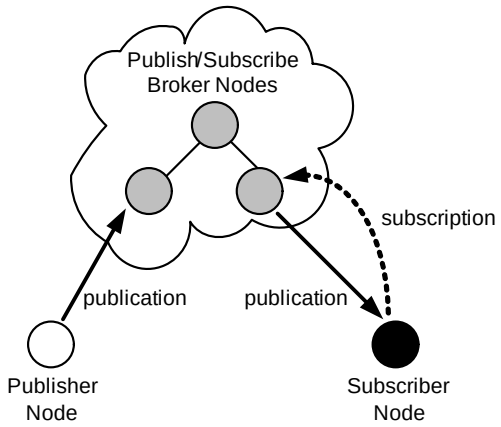


Figure 1.1: Publish/Subscribe system

brokers match and route relevant publications to interested subscribers. The messages in a content-based pub/sub system are routed based on their content. Therefore, publishers and subscribers are loosely coupled and require no knowledge about each other in order to communicate. This feature is essential to distributed BPEL orchestration engines, which coordinate the execution of loosely coupled services. Moreover, business workflow tasks or jobs can be executed by job execution agents, which are typical pub/sub clients connected to brokers. The job execution agents are lightweight components as they have no specific logic for workflow management. After the task or job information has been deployed (using appropriate subscriptions) to job execution agents, they only need to receive publications that triggers their execution, so as to process the corresponding tasks or jobs, and send the produced publications.

Based on the above idea we propose a distributed BPEL orchestration engine implemented over a content-based pub/sub system. On the top level, our engine must support the execution of business processes specified in BPEL[8]. The BPEL processes consist of activities divided into two categories: basic and structured. The supported basic activities define synchronous and asynchronous communication with outside Web services (receive, reply, invoke), control execution of business processes (exit, wait, empty), and update data values of variables (assign). The supported structured activities are used to express typical and complicated workflow patterns: sequence (by sequence), exclusive choice (pick), multiple-choice (if/else), parallel split (flow), and repetition (while).

In order to support the distributed orchestration of the aforementioned BPEL activities, our distributed execution engine provides four basic operations:

1. BPEL *process description parsing and decomposition*. The BPEL process is parsed into basic and structured activities and all the activities are transformed into pub/sub messages. Thus, this step provides the mapping from BPEL to pub/sub semantics.

[8]We provide support for only a subset of the original BPEL activities.

2. *Activity deployment.* The pub/sub messages produced by the previous operation are sent to specific agents of the distributed infrastructure using network proximity and node utilization metrics.

3. *Activity execution.* The agents execute the deployed activities and publish new pub/sub messages.

4. *Activity undeployment.* The engine removes the deployed activities from the agents.

The above operations are implemented using an underlying pub/sub infrastructure. This is facilitated by a structured P2P network overlay based on *MSPastry* [CCR04], where the job execution agents are assigned to network nodes. *MSPastry* offers infrastructureless scalability (thus requires no specialized administration needs[9]), provides message propagation based on network proximity, has fault-tolerance properties (as every node has equal role and there are no single points of failure), but offers only exact name lookups. Thus, we extend this infrastructure by implementing content-based pub/sub semantics on top of the Distributed Hash Table (DHT) interface using efficient algorithms for storing, matching, and delivering pub/sub messages. With our scheme we express the dependencies among the business processes and the agents using pub/sub messages. By exploiting all these properties our distributed engine has the ability to: a) support scalable deployment and execution of BPEL processes, b) dynamically adapt the processes deployment and execution based on the network status without any human intervention, and c) provide good fault tolerance even in a network with high churn rate.

1.3 Contributions

The main contributions of this thesis are:

- *An infrastructureless network based on multiple-purpose agents.* We propose a distributed BPEL orchestration engine architecture based on a content-based pub/sub system that is implemented on top of a DHT network. Our engine is inspired by NIÑOS [LMJ10], but uses a DHT network that consists of multiple-purpose agent nodes and requires no human intervention to shape itself efficiently.

- *Two dynamic activity mapping mechanisms based on the network utilization.* Our engine support flexible mapping of the BPEL activities to the engine's nodes, thus

[9] On the contrary, the traditional distributed pub/sub systems require an application-level overlay network, where its configuration is typically performed manually by a network deployer and plays an important role in its performance. This task becomes more difficult as the number of brokers in the network scales up. Furthermore, the topology is often static and does not adapt to changing usage patterns. In addition, the commonly organization of the broker network as a tree results in every node being a single point of failure or a bottleneck. Therefore, many existing distributed pub/sub systems have good scalability properties but require the purchase of specialized infrastructure and administration.

permitting the system to shape itself based on the network utilization state. To this end, we propose and evaluate two deployment mechanisms with different characteristics.

- *A* pub/sub *dissemination algorithm with proximity awareness.* We extend and modify the pub/sub algorithms that where original proposed in Ferry [ZH05]. We use pub/-sub matching techniques that reduce the size of the nodes' routing tables. Furthermore, by exploiting the DHT layer proximity properties we provide proximity-aware subscription installation and event propagation, yielding good message delivery performance.

- *A quantitative performance evaluation of the proposed engine.* An experimental evaluation is carried out. The evaluation shows that the pub/sub mechanism scales well and produces low network overhead. Furthermore, our engine's performance is evaluated versus centralized clustered solutions, based on the exhibited average execution time, throughput, and overhead.

1.4 Thesis Outline

The rest of the thesis is organized as follows:

- Chapter 2 presents an overview of the related theory and technologies. Readers not relevant with the discussed areas are recommended to read this chapter, while others can freely skip the presented material.

  It starts by presenting basic concepts like: business process, workflow, workflow management systems, and Web services. Next, it presents the notion of business process management and makes a short introduction of the BPEL language constructs and execution environment. It continues with the pub/sub model and emphasizes in the distributed content-based pub/sub. Last, it presents the most popular structured P2P architectures based on distributed hash tables such as *MSPastry*, the overlay used in our work.

- Chapter 3 discusses related work in the field of pub/sub systems and the distributed orchestration engines. It starts with content-based pub/sub systems that use different overlay and models, while it focus on systems based on the DHT infrastructure, like Ferry. Then, it presents related decentralized orchestration engines like NIÑOS, with special focus on their execution mechanism.

- Chapter 4 discusses the approach taken by this thesis to the problem of decentralized business process execution and provides details about our implementation. It starts by providing an overview of the architecture and its constituting elements and presents the pub/sub model that is employed by our solution. Then, it presents the proposed mapping between pub/sub and the BPEL language constructs. Finally, it

concludes with a detailed description of the engine's operation during the process deployment, execution, redeployment, and the undeployment phase.

- Chapter 5 discusses the evaluation methodology followed in this thesis, and provides comments on its results. It starts with the *PeerSim* simulation engine [MJ09] that was used to implement our proposed architecture and presents both its benefits and drawbacks. Next, it presents an evaluation of the used pub/sub mechanisms. We demonstrate that our pub/sub algorithms scale well and have good load balancing features. In the following, it presents an evaluation of the engine's performance in terms of process execution time, throughput, and overhead. Also, presents a performance comparison between our solution and a clustered centralized engine. Finally, concludes with the comparison of the two proposed deployment mechanisms (i.e per-process and per-instance deployment).

- Chapter 6 concludes the thesis, summarizes, and identifies various directions and open problems for future work.

- Finally, Appendix A presents the detailed mapping of the BPEL activities to the used pub/sub language.

Chapter 2

Background

This chapter starts with Section 2.1 which introduce the reader into the basic concepts of business process, workflow, and workflow management systems. Then, in Section 2.2, we explain the notion of Web services as they are central to our work.

Section 2.3 presents the notion of business process management and makes a short introduction of the BPEL language constructs and execution environment. BPEL is the language that our work is based upon and has a main role into the orchestration of Web services.

Second, Section 2.4 presents the pub/sub model and its categorization on different types based on the used architecture and filtering method. We emphasize particularly in the distributed content-based pub/sub as it is used in our work.

Last, Section 2.5 makes a short introduction to the ideas that drove the evolution of the peer-to-peer systems and the benefits that this approach offers to the distributed computing. We use a common categorization and divide our presentation into unstructured and structured peer-to-peer networks. The latter are related to our work, so we present the most popular structured architectures based on distributed hash tables. We analyze the operation of five prominent DHT systems and discuss their benefits and drawbacks. We conclude this section with the presentation of *MSPastry*, the overlay used in our work.

2.1 Workflow Management

A *process* (or *procedure*) is a collection of related tasks that need to be carried out in an order determined by a set of conditions. Its main goal is to produce a specific service or product for a particular customer or customers.

A task is a logical unit of work that is carried out as a single whole by one resource and needs to be accomplished within a defined period of time. A resource is the generic name for a person, machine, or group of persons/machines that can perform specific tasks. The resource does not always carry the task independently, but it is always responsible for it. A *business process* is a process specialized in the domain of business

organisational structure and policy. Its purpose is to achieve business objectives.

Figure 2.1 presents an example of a business process, where an insurance company deals with a claim. At first it receives the client's request and establishes its type (e.g. a flood request). Notice that these activities are executed in sequence. Then the process checks in parallel the client's profile and his policy to confirm that she/he is a valid customer and has insurance coverage for the particular claim. Based on the output of the previous activities, the rejection activity makes a selection and either produces a rejection letter or calculates the size of the payment that is offered to the client. The client may reject the offer; in this case an iterative assessment of an objection can take place, until a client agrees with the payment size and a settlement is made. Finally, the client is paid with the offered amount.

A *workflow* is the computerised facilitation or automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules [Coa99].

Workflows have been applied with success in telecommunications, software engineering, banking and financial industry, manufacturing and shipping, health and sanitary, office automation and scientific research fields such as bioinformatics, cheminformatics, ecoinformatics, geoinformatics, and physics. For example, Figure 2.2 is an example of a radiology workflow containing patient registration, appointment scheduling, examination performing, medical reporting, and data/image archiving. Moreover, significant effort has been put into defining workflow patterns that can be used to compare and contrast different workflow engines, even across different domains. This process resulted into the definition of the workflow patterns summarized in work of Van Der Aalst et al. [AHKB03]. The authors provided independence from specific workflow languages and addressed business requirements in an imperative workflow style expression. In essence, these patterns allowed a potential mapping to be positioned closely to different languages and implementation solutions.

A *workflow management system* (WfMS) is a system that defines, creates, and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications [Coa99].

A workflow management system can be compared with an operating system: it controls the workflows between the various resources, people, or applications. It is confined to the logistics of case handling. In other words, a change to the content of case data is implemented only by people or application programs. A workflow management system has a number of functions that can be used to define and graphically track workflows, thus making both the progress of a case through a workflow and the structure of the flow itself easy to revise. Figure 2.3 illustrates the basic characteristics of a WfMS its basic functions [Hol95]. These belong to three types: a) build-time functions, concerned with defining the workflow process and its activities, b) run-time control functions, concerned with managing the workflow processes, and c) run-time interaction functions, concerned with human users and IT-tools.

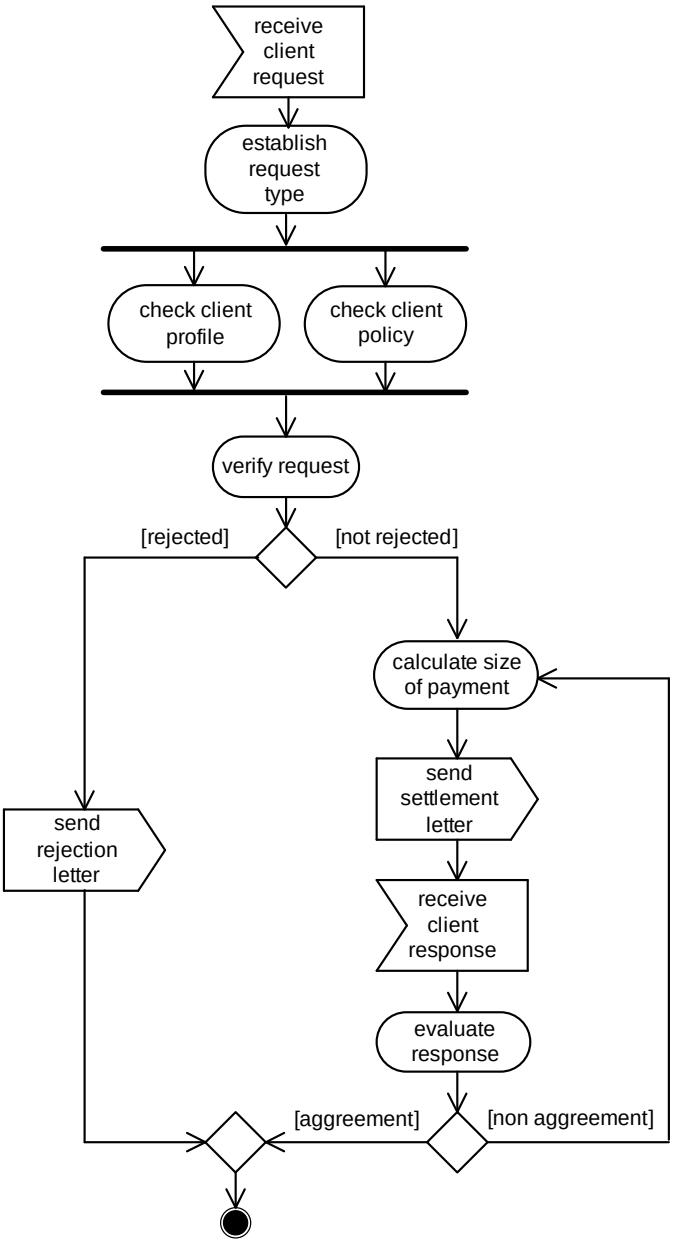


Figure 2.1: Example of an insurance claim business process

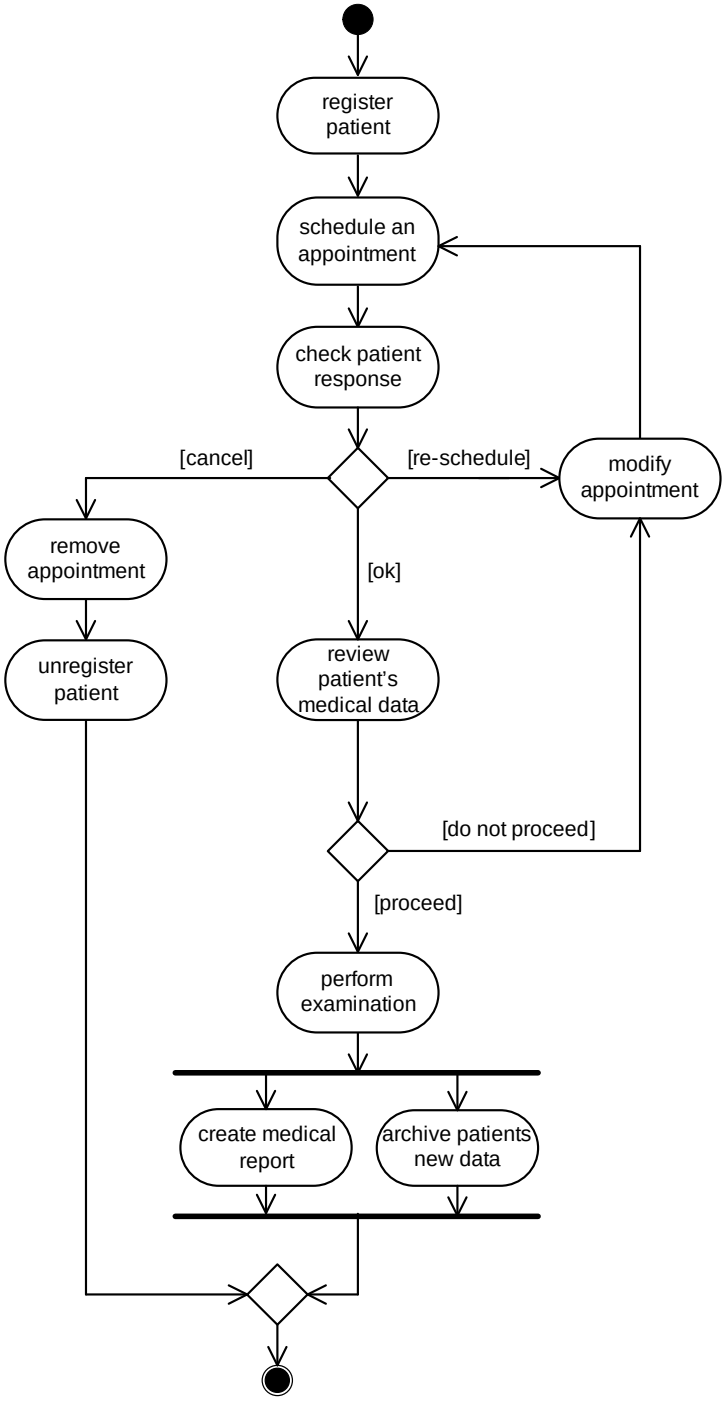


Figure 2.2: Example of a medical workflow

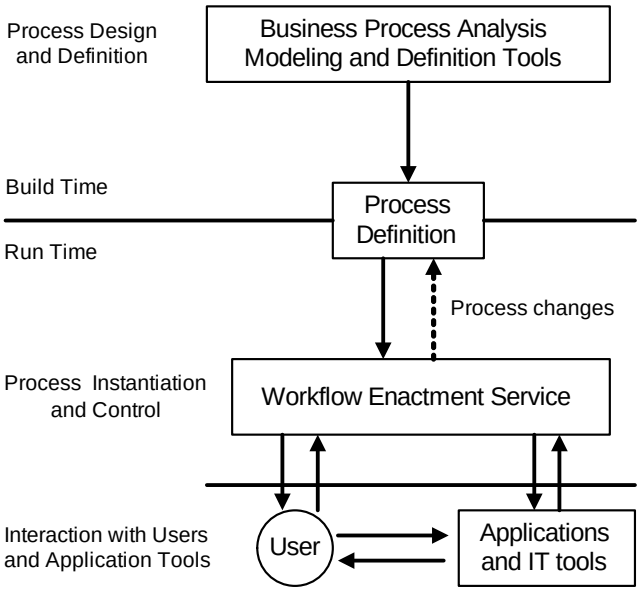


Figure 2.3: Workflow system characteristics

2.2 Web Services

Web services provide interoperability among applications using different software platforms, operating systems, and programming languages. A web service is a distributed application whose components can be deployed and executed on distinct devices. For instance, a stock-picking web service might consist of several code components, each hosted on a separate business-grade server, and the web service might be consumed on PCs, handhelds, and other devices.

Web services are based on the following standards: a) eXtensible Markup Language (XML) [BPSM+06] as a common definition language, b) Web Service Description Language (WSDL) [CCMW01] as a common format for defining interfaces, c) Universal Description, Discovery and Integration (UDDI) [CHvRR04] to define how to publish and discover services, d) Simple Object Access Protocol (SOAP) [BEK+00] as a common format for the messages sent between software components, and e) Hyper Text Transport Protocol (HTTP) [FGM+97] as the delivery mechanism. The proliferation of these standards reflects the demand for distributed enterprise applications to communicate with software services provided by vendors and clients.

As shown in Figure 2.4, the Web Services' Model consists of basic operations such as describe, publish, discover, bind, invoke, update, and unpublish. The Service Provider is an individual (organization) that provides services. The Service Provider's job is to create, publish, maintain, and unpublish services. From a business point of view, the Service Provider is the owner of the service, whereas from an architectural view, it is a platform which holds the implementation of the service. The Service Broker provides a

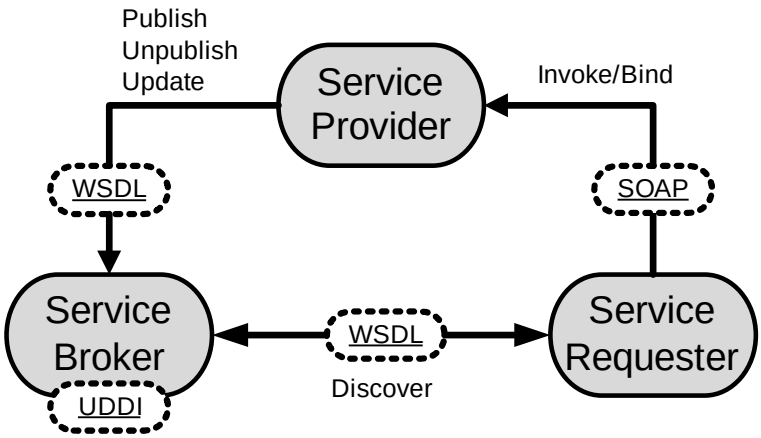


Figure 2.4: Web services model

repository of service descriptions (WSDL). These descriptions are published by the service provider. Service Requesters will search the repository to identify the needed services, and obtain the binding information for these services. A service broker can either be public, where the services are universally accessible, or private, where only specified sets of Service Requesters are able to access the service. The Service Requester is a party that looks for a service to fulfill its requirements. A requester can either be a human accessing the service, or an application program (the program could also be another service). From a business view, the Service Requester is a business that wants to consume a particular service, whereas from an architectural view, it is an application that looks for and invokes a service.

Web services can be divided roughly into two groups: SOAP-based and REST-style. In SOAP-based web services, the SOAP is the underlying infrastructure where the service's requests and the corresponding responses are exchanged via XML based messages. Both SMTP and HTTP are valid application layer protocols used to transport SOAP messages. Nevertheless, HTTP has gained wider acceptance as it works well with today's Internet infrastructure.

REST stands for REpresentational State Transfer and was proposed in Roy's Fielding Ph.D. dissertation [Fie00], to describe an architectural style in the design of Web services. While SOAP has standards [BEK+00], toolkits, and bountiful software libraries, REST has no standards, few toolkits, and meager software libraries. The REST style is often seen as an antidote to the creeping complexity of SOAP-based Web services. REST was initially described in the context of HTTP, but is not limited to that protocol.

REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a

resource. At any particular time, a client can either be in transition between application states or "at rest". A client in a rest state is able to interact with its user, but creates no load and consumes no per-client storage on the servers or on the network. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be in transition. The representation of each application state contains links that may be used next time the client chooses to initiate a new state transition.

In short, the Web services have the following pros:

- Exhibit loose coupling, while the operations exchange data and not state.
- Their operations are based on XML based input, output, and fault messages. The combination of the messages defines the type of the operation (one-way, request-response, solicit-response, or notification).
- Provide asynchronous and synchronous interactions.
- They are stateless.
- Use common protocols such as HTTP, SMTP, FTP, and MIME.

On the other hand, the Web services have the following cons:

- Non-RESTful Web services are often considered too complex.
- They exhibit lesser performance than the binary protocols, due to the use of XML as a message format and SOAP/HTTP in enveloping and transport.
- There is no inherent provision of Quality of Service (QoS), security, or transaction processing.

2.3 Business Process Management

SOA provides a powerful structure for business process management by comprising loosely coupled and highly interoperable application services. Furthermore, with the development and maturity of Web services, people have found a suitable technical foundation for making business processes accessible within the same enterprise and across different enterprise domains. This way Web services provide access to operations of certain applications and information systems.

Business processes are designed top-down with starting points and ending points, they may be repeatable, and they use Web services. Furthermore, each business process is exposed as a Web service. In this regard, the role of Web Services Business Process Execution Language (WS-BPEL or BPEL) [AAA+07] is extremely significant. BPEL is a language for defining and executing business processes using Web services and supports numerous workflow patterns that are presented in Table 2.1. BPEL has attained broader

acceptance in the industry since its first version was developed in 2003 [ACD+03] and enables the realization of SOA through the composition, orchestration, and coordination of Web services. With BPEL, enterprises standardize the way to define their business processes. In turn, this leads to business process optimization, re-engineering, and the selection of the most appropriate processes, thus further optimizing the organization. Based on this advantage, more and more vendors have developed a complete BPEL engine as a necessary core part in their flagship SOA products, including Oracle[1], IBM[2], and Microsoft[3].

BPEL provides enough features to let the user define complex business processes in an algorithmic manner. The user can declare variables, express conditional behaviors, construct loops, define fault and compensation handlers, parallelize multiple operations, and so on. The fundamental units in a BPEL business process are the activities, which are classified into two types: basic activities and structured activities. Basic activities represent primitive constructs and are used for common tasks such as invoking Web services, while structured activities define the control flow. Table 2.2 lists the most important basic and structured activities as defined in BPEL. With the combination of basic and structured activities the designer of a BPEL process can form complex algorithms that specify exactly the steps of business processes.

In traditional BPEL engines, Web services can be combined in two different ways: orchestration or choreography. In orchestration (as depicted in Figure 2.5(a)), there is a centralized coordinator that controls the involved Web services. The explicit definitions of operations and the order of invocation of Web services are given to the central coordinator. The involved Web services do not need to know each other if they are involved in a composition process and are part of a higher business process. For this reason, orchestration is mainly used in private business processes. Contrary, as illustrated by Figure 2.5(b), choreography does not rely on a central coordinator. Every Web service knows exactly when to execute its operation and with whom to interact with. All participants need to be aware of the business process, the interactive partners, the messages to exchange, and the time of executing operations. Thus, the choreography is a collaborative effort focused on the exchange of messages in public business processes.

Both orchestration and choreography are supported in BPEL through describing business processes in two different ways: executable processes and abstract business processes. Executable processes specify the details of business processes and can be executed by an orchestration engine, while abstract business processes do not include the internal details of processes and are not executable. To conclude, orchestration is a more flexible paradigm, although the line between orchestration and choreography is vanishing. In essence, orchestration has the following advantages:

- There is exactly one entity responsible for the execution of the whole business process.

[1] http://www.oracle.com/technetwork/middleware/weblogic/
[2] http://www.ibm.com/software/websphere/
[3] http://www.microsoft.com/biztalk/

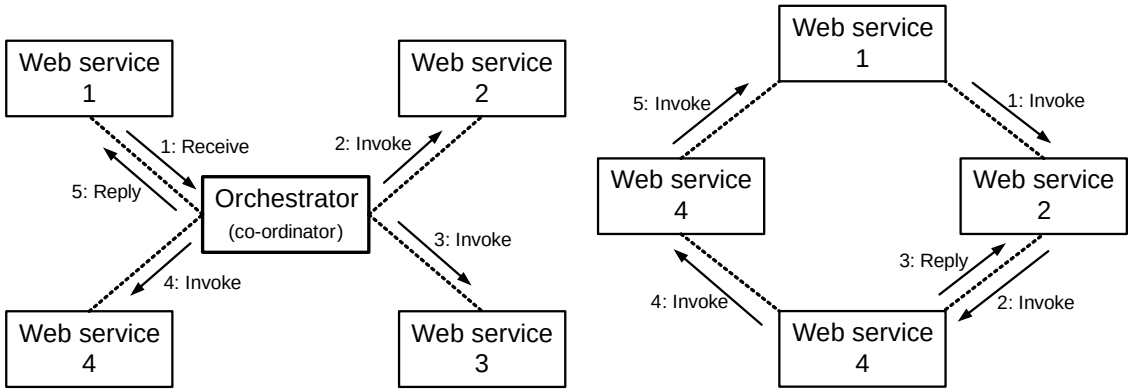
Class	**Pattern Name**	**Pattern Description**
Basic Control	Sequence	Execute activities in sequence
	Parallel Split	Execute activities in parallel
	Synchronization	Synchronize two parallel executions
	Exclusive Choice	Choose only one execution path
	Simple Merge	Merge two alternative execution paths
Advanced Branching	Multiple Choice	Choose several execution paths
	Synchronizing Merge	Merge many execution paths
Structural	Implicit Termination	Terminate if there is nothing to be done
Patterns Involving Multiple Instances	MI without synchronization	Generate many instances of one activity without synchronizing them later
	MI with a priori known design time knowledge	Generate many instances of one activity when the number of instance is known at the design time
State-based	Deferred Choice	Execute one of the two threads
	Interleaved Parallel Routing	Execute two activities in random order, but not in parallel
Cancellation	Cancel Activity	Disable an enabled activity
	Cancel Case	Cancel the process

Table 2.1: BPEL supported workflow patterns

Basic Activities	
Activity	**Description**
receive	Blocking wait for a message to arrive
reply	Respond to a synchronous operation
assign	Manipulate state variables
invoke	Synchronous or asynchronous Web service call
wait	Delay execution for a duration or deadline
throw	Indicate a fault or exception
compensate	Handle a fault or exception
exit	Terminate a process instance

Structured Activities	
Activity	**Description**
sequence	Sequential execution of a set of activities
scope	Partitions process into logically organized sections
compensate	Undones a previously-completed unit of work
if	Conditional execution based on instance state
while	Looping construct
repeatUntil	Looping construct
forEach	Looping construct
pick	Conditional execution based on events
flow	Concurrent execution

Table 2.2: BPEL basic and structured activities



(a) Web services orchestration (b) Web services choreography

Figure 2.5: In the orchestration there is a centralized coordinator that controls the involved Web services, as shown in Figure 2.5(a). Contrary, in the choreography every Web service knows when to execute its operation, as shown in Figure 2.5(b).

- Web services can be incorporated, even when they are not aware that are a part of a business process.
- When faults occur alternative scenarios can be provided.

2.4 Publish/Subscribe Paradigm

Three entities are involved in the publish/subscribe (pub/sub) paradigm: producers, consumers, and brokers. Information producers submit data (using publications) to the system and information consumers indicate their interests by submitting subscriptions. Subscriptions have a notification set, which is the set of potential publications that would match the subscription. On receiving a publication, a broker determines which subscriptions match the publication and forwards notifications to the appropriate subscribers. The pub/sub paradigm has recently become quite popular in both research [CRW01, CDNF01, MÖ1, OAA+00, PB02] and commercial communities, finding widespread use in applications ranging from selective information dissemination to network and distributed system management.

The first criterion for the categorization of the pub/sub systems is based on their architecture: centralized or distributed. pub/sub was first implemented in centralized client-server systems, such as Elvin [SA97]. Elvin uses a central server that stores all the subscriptions, evaluates the subscriptions upon events, and delivers events to the matched subscribers. Centralized solutions, while simple, have an inherent scalability problem as the number of events and subscriptions in the system increases. For this reason, current research focuses mainly on distributed pub/sub, which provides natural decoupling of publishers and subscribers. Since publishers are unconcerned with the potential consumers of their data and subscribers are unconcerned with the locations

of the potential producers of interesting data, the client interfaces of a pub/sub system are simple and intuitive.

Distributed pub/sub systems [BCM+99, CRW01, TE04, CW03, CRW04, CS05, BCM+99] typically contain a network of interconnected brokers, each providing client binding interfaces for publishers and subscribers. Publishers and subscribers are considered clients of the distributed pub/sub system. The brokers act as network servers, providing pub/sub messaging services to the clients. Typically, in this case a Peer-to-Peer (P2P) network is used, where the brokers are organized as super-or ultra-peers and the clients (publishers and subscribers) are organized as leaf-peers[4].

In the pub/sub model, subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called filtering. Based on the used filtering method, the pub/sub systems can be categorized into four different classes: a) subject-based, b) topic-based, c) type-based, and d) content-based systems.

In subject-based pub/sub systems systems, such as the Information Bus [OPSS93], Bay-eux [ZZJ+01], and Scribe [RKCD01], the producers publish notifications with respect to a certain subject. For example, in a subject-based system for stock trading, a participant could select one or two stocks and subscribe based on stock name, if that were one of valid subscription fields. A drawback of this filtering method is that the user receives all events that are associated with that subject (stock name). Thus, if a participant was interested only for the P/E-ratio[5] (price-to-earnings ratio) of a stock, she/he would likely receive much more information than needed.

Topic-based pub/sub systems extended the subject-based model by allowing a hierarchy of topics. Each publisher and subscriber joins the groups containing the topics in which they are interested and events that belong to a topic are broadcast to all subscribers of the corresponding group [Bir93, OPSS93].

Type-based pub/sub systems are a variation on topic-based systems in which publications have a type [EGD01]. The type concept is similar to C++ classes, in that it signifies the structure or contents of the publication data. The types are also potentially hierarchically organized, as with super- and sub-classes. Subscriptions indicate the desired type of publications.

Content-based pub/sub systems are preferable as they allow subscribers to specify their interests in a fine-grained way, as they can specify constraints on the actual data within the publication. There is typically no hierarchy of content-based messages. The content-based pub/sub scheme is defined as $S = \{A_1, A_2, \ldots, A_n\}$, where each A_i corresponds to an attribute. Each attribute has a unique name, type, and domain and can be specified by a quadruple [*name, type, min, max*]. The *type* could be integer, float, string, etc. The *min* and *max* define the range of domain values taken by the given attribute. An event is a set of equalities over the attributes $\in S$ and it can be represented as

[4]For the definition of these terms please refer to Section 2.5

[5]P/E ratio is a measure of the price paid for a share, relative to the annual net income (or profit) earned by the firm per share.

$e = \{A_1 = c_1, A_2 = c_2, \ldots, A_n = c_n\}$. In general, events may specify values for a subset of the attributes, for example $e_i = \{A_1 = v_1, A_3 = v_3\}$.

A predicate is used to specify a constant value ($=$) or range ($<, \leq, >, \geq$) for an attribute, and is specified by a quadruple [*name, type, operator, value*]. A subscription is a conjunction of predicates over one or more attributes. If a subscription needs to specify multiple predicates over the same attribute, it can be modeled as a combination of multiple subscriptions, each of which specifies one value or continuous range over the attribute. For simplicity, in our presentation we assume that each subscription specifies a value or continuous range over attributes. An example of subscription is $s_i = (A_1 = v_1) \wedge (v_2 \leq A_3 \leq v_3)$. An event e matches a subscription s if each predicate of s is satisfied by the value of the corresponding attribute contained in e.

The increase in expressiveness allows the delivery of uninteresting notifications to be reduced or even to be avoided. Moreover, only content-based selection provides full decoupling of producers and consumers, facilitating extensibility and continual change. Clearly, content-based filtering is the most interesting notification filtering mechanism, but on the other hand, scalable implementations are the most complex to realize, too. Indeed, the expressiveness of the selection predicates that can be applied has a large impact on the scalability of any content-based notification service. In the literature, several systems relying on content-based selection are described. Representative examples are: Gryphon [BCM+99], SIENA [CRW01], Narada [FPR02], Le Subscribe [PFL+00], JEDI [CDNF01], CEA [BHM+01], Hermes [PB02], and REBECA [MÖ1].

2.5 Peer-to-Peer Systems

Peer-to-Peer (P2P) networks were designed to overcome some of the limitations imposed by the client/server model. The nodes in a P2P network provide both server and client services and contribute resources, which may include sharable content, bandwidth, storage space, and computing power to the overall system operation. For this reason, the term peer is used to define a network node that acts both as client and server. The peers create virtual or logical links with their neighbor peers thus creating an overlay network, which is built on top of the physical network. This overlay forms an application level network topology that does not necessarily correspond to the physical network topology (i.e. each virtual link may correspond to a path, perhaps through many physical links, in the underlying network).

As more peers join in the P2P network, there is an increase on the demand but also on the overall capacity of the system; as the peers are characterized by sharing their computer resources by direct exchange, more resources are available to benefit all users. This property of the P2P networks is commonly referred as organic scaling [HHL+03]: the aggregate resources in the network grows naturally with the application utilization. In contrast, in a typical client-server architecture, clients share only their demands with the system, but not their resources. In this case, as more clients join the system, less resources are available to serve each client. The distributed nature of P2P networks

also increases robustness, especially if they exhibit built-in fault-tolerance, replication, and load balancing features. By enabling peers to find the data without relying on any form of centralized index servers (in pure P2P systems[6]) and due to geographically dispersed content replication, there is no single point of failure in the system. Moreover, in P2P architectures the nodes are autonomous and self-organized; while in client/server model the servers are under some administrative authority, in a P2P architecture the peers are under no administrative control.

P2P networks are dynamic and peers can be extremely transient [GDS+03], entering and leaving the network frequently, while in the client/server model the servers usually enjoy long uptimes. For this reason P2P networks are designed to treat instability and variable connectivity as the norm, automatically adapting to failures in both network connections and computers, as well to a transient population of nodes. On the other hand, the nodes in a P2P network exhibit increased heterogeneity as they usually have widely varying capabilities. These include wide differences in network bandwidth, processing power, memory, and storage, that complicate the efforts for achieving load-balancing and fault-tolerance.

We must state that the P2P architecture is not an attempt to replace the client/server model. There are many applications for which each model is more appropriate. For example, the client/server model is probably more appropriate for applications that require some administrative control, or those that do not require very large scalability. P2P networks can be loosely classified as unstructured and structured [ATS04]. The following sections will discuss these classes in more detail.

2.5.1 Unstructured Networks

Unstructured P2P networks have an overlay topology built in an ad-hoc fashion, with randomly established links. Such networks are created when new nodes join into random network positions, copy the neighbor links of already connected nodes, and afterwards create their own connections. In this type of network, when a node searches for a particular type of information, a hop-limited flood query is imposed, in order to find as many nodes that share and provide the requested content. The basic disadvantage of this type of network is that the query is not always satisfied, because there exist no guarantee that a limited flood search will find a peer with the requested content. Furthermore, this search mechanism exhibits high overhead, thus causing low search efficiency. Examples of unstructured networks are Napster [Nap] and Gnutella [Gnu]. These networks have been used by large scale file sharing applications and differ mainly on the employed searching mechanism.

Napster uses a centralized index server which holds a metadata list about the files shared by the users. In order to share a file, a user must upload the file's description on the central server. The latter binds the file's metadata with the user's network

[6] totally distributed systems, in which all nodes are completely equivalent in terms of functionality and the tasks they perform.

address. After the completion of this process any user can query the server for the specific file. The server matches the query with all the metadata lists and replies with the network addresses of the peers that host the matching files. This way, requesters can download these files directly from the sharers. Napster is characterized by a centralized (client/server) search, but P2P file transfer[7]. The advantage of a centralized search server is that query matching can be ensured. Nevertheless, the centralized index is a single point of failure and can become a scalability bottleneck.

In Gnutella, every peer has an equal role and the searching mechanism is completely decentralized. The network topology is created ad-hoc with no imposed structure. In order to search for a file, a requester must flood its neighbors with a query. Then they repeat the process and also flood their neighbors. As the query message has a a time-to-live (TTL) field that is reduced in each propagation, the network is protected from excessive messages by dropping all messages with TTL=0. At some point, a couple of peers which host the requested file are reached. These peers reply to the requester and the latter downloads the file directly from one of these providers. The distributed search does not suffer from a single point of failure, but can no longer ensure that all matching files will be found. Also, the searching mechanism is not scalable as a single query can result in many messages, congesting the network with unnecessary traffic.

2.5.2 Structured Networks

The P2P networks based on distributed hash tables (DHTs) are called structured P2P networks. A DHT is a distributed version of a hash table. It stores a (key, value) pair at a network node that is deterministically computed using the key. Hence, any node can retrieve the stored value based solely on the key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, namely, each peer in the network stores a subset of the (key, value) pairs in the system. This way, a change in the set of participants causes a minimal amount of disruption. This, in turn, allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures in a self-organizing manner. Furthermore, DHTs provide theoretical performance guarantees for: a) the balanced key distribution among the peers, b) the expected routing state size that a peer is responsible to maintain, and c) the required number of hops to lookup a key in the DHT.

Early DHTs were designed to provide distributed file storage. In these implementations the filename is used as key, and the file contents are used for the value. Nevertheless, both the key and and the value may be an arbitrary string of bytes. These early implementations were followed by a surprising number of diverse distributed applications built on top of the DHT interface, including data streaming [CDK+03, BB05, HFC+08], co-operative messaging [MPR+03, SMK08], storage systems [RD01a, DKK+01], name servers [RS04], file sharing [NWD03], databases [HHL+03], and Inter-

[7]Napster is commonly categorized as a hybrid application, because it combines a P2P network with a centralized index.

net telephony [SS05].

A DHT implementation is an application layer (OSI layer 7) protocol that provides an Application Programming Interface (API) for other applications. The most interesting DHT implementations are categorized into three mechanisms for routing messages and locating data:

Chord is a system whose nodes are mapped in an identifier table and maintain a distributed routing table in a form of an associated finger table.

CAN is a system which uses a n-dimensional Cartesian coordinate space to implement the distributed location and routing table, while each node is responsible for a zone in the coordinate space.

Pastry (also Tapestry and *MSPastry*) is a system based on the plaxton mesh data structure [PRR97], which maintains pointers to nodes in the network whose IDs match the elements of a tree-like structure of ID prefixes up to a digit position.

To illustrate the DHT operation, we will use the Pastry [RD01b] DHT implementation. Other DHTs have similar interface with Pastry, but differ in the implementations and performance. Nevertheless, the core operation of every DHT protocol is to map a key to a node and efficiently route messages to this node. Thus the DHT interface consists of the insert() and lookup() functions (in Pastry they are called route() and deliver() respectively) as presented in Table 2.3. In the discussion below, we assume a DHT network of N nodes and K keys.

Identifier circle

Each peer in the DHT network is assigned a 128-bit node identifier (nodeId). Furthermore, in every (key, value) pair stored in the DHT, the key is a 128-bit number, while the value is a sequence of bytes. The nodeIds and the keys are a sequence of digits with base 2^b (where b is a parameter that is typically equal to 4) and belong to a circular 128-bit identifier space, which ranges from 0 to $2^{128} - 1$. The nodeIds and the keys are generated by the SHA-1 [EJ01] cryptographic hash function to ensure that they are uniformly distributed in the 128-bit identifier space. As a result of this random assignment, there is high probability that nearby nodes in the identifier circle have distant placement in the network topology and in their geographical locations.

Key assignment

As stated before, the cryptographic hash function distributes keys and nodeIds evenly around the identifier circle. Thus no node stores a disproportionate share of the (key, value) pairs in the system. With high probability, in a network with N nodes and K keys, each node stores K/N pairs. The method used by Pastry to map keys into the identifier circle places the keys to the node with the numerically closest nodeId, as show by the example in Figure 2.6.

Basic DHT operations	
Operation	**Result**
insert(key, value)	Stores (key,value) into the node responsible for the key.
lookup(key)	Retrieves the value associated with the key from the appropriate node.

Pastry DHT operations	
Operation	**Result**
route(key, msg)	Route the given message to the node with nodeId numerically closest to the key.
deliver(msg)	Called when a message is received and the local node's nodeId is numerically closest to key, among all live nodes.

Table 2.3: Basic DHT operations and specific Pastry DHT operations

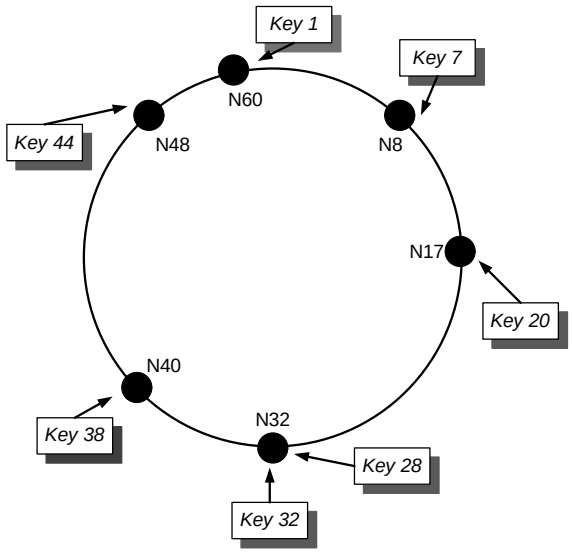
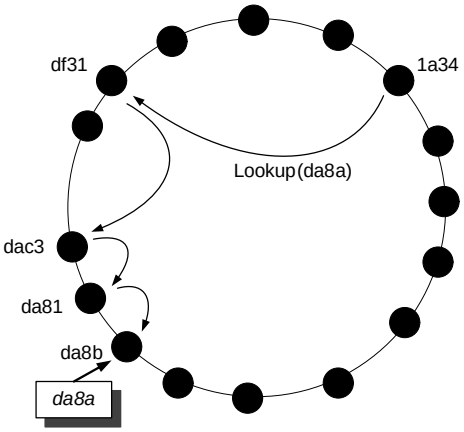


Figure 2.6: Node and key assignment in an identifier space with range [0,63]

Routing algorithm

Pastry routes each message to the node whose nodeId is numerically closest to the given key. Pastry uses a prefix routing algorithm where at each hop the message is sent to a node that matches the destination nodeId by at least one more digit (each digit is b bits). A hop refers to a message sent from one node to another using a DHT overlay link that may correspond to multiple physical network links. If no such node is found, then the message is forwarded to a node that is numerically closer to the destination nodeId than the current node's nodeId. Figure 2.7(a) shows an example of message routing. Pastry's routing algorithm is able to route a message to the destination in less than $\lceil log_{2^b} N \rceil$ hops under normal operation, while similar performance guarantees are typical in all DHT implementations.



(a) Pastry prefix routing

NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

(b) Pastry node state

Figure 2.7: Pastry prefix routing mechanism is shown in Figure 2.7(a). Each node stores state about its leaf set, routing table, and neighborhood set as shown in Figure 2.7(b).

Node state

Each Pastry node maintains a routing table (R), a neighbourhood set (M), and a leaf set (L), as shown if Figure 2.7(b). The routing table, R, stores various nodeIds in a table with $\lceil log_{2^b} N \rceil$ rows and $2^b - 1$ columns. The entry in the ith row and jth column has the first i digits in common with the current node's nodeId, while the jth digit has the value j. Each entry in R also contains the IP address of one of the potentially many nodes whose nodeId has the appropriate prefix. As several nodeIds meet the constraints of an entry in R, Pastry chooses the nodeId according to a proximity meter. Typically, the number of physical hops is used to determine the proximity between nodes. The value of b can be used to trade off routing table size (about $\lceil log_{2^b} N \rceil * (2^b - 1)$ entries) and routing hops ($\lceil log_{2^b} N \rceil$).

The neighbourhood set, M, stores the nodeIds and IP addresses of the $|M|$ closest nodes (according to the proximity metric) to the current node. $|M|$ is typically $2 \times (2^b)$

(e.g. 32 or 64). While the neighbourhood set is not required for correct routing, it is used to maintain locality properties in order to optimize routing.

The leaf set, L, stores the nodeIds and IP address of the $|L|/2$ numerically closest nodes with a nodeId larger than the current node in the identifier circle, and the $|L|/2$ nodes with nodeId smaller than the current node. $|L|$ is typically 2^b (e.g. 16 or 32). During routing, if the destination nodeId falls within the leaf set, the message is sent directly to the appropriate leaf node bypassing prefix routing, as described above.

Self-organization and adaptation

Pastry nodes automatically update their state, as nodes arrive and depart. A node entering the network must know the IP address of an existing node in the overlay network; the state of this node is used to bootstrap the entering node. In Pastry, to maintain good performance, it is desirable for the known node to be close (according to a proximity metric) to the entering node. This is necessary to ensure that a node's routing state is initialized with nearby nodes. An entering node requires $O(log_{2^b}N)$ messages to populate its state. Node departures or failures can cause entries in the routing table to become unreachable. In this case, a node finds a replacement entry by querying other nodes in its routing table.

Fault-tolerance

Pastry guarantees eventual delivery of a message, unless $\lfloor |L|/2 \rfloor$ nodes with adjacent nodeIds fail concurrently. Note that the routing guarantee of $O(logN)$ hops is not guaranteed when failures occur.

2.5.3 DHT implementations

Chord [SMLN+03] in a similar vein uses a circular identifier circle, while the nodes are identified by keys. The keys are assigned both to files and nodes by means of a deterministic function [KLL+97]. A minor difference is that Chord stores a key at the first successor node in the identifier circle (i.e. the first node whose identifier is equal to, or follows k, in the identifier space), as illustrated in Figure 2.8. Each Chord node n maintains a finger table of m entries, where m is the number of digits in a node identifier. In this table, each entry i points to the successor of node $n+2i$. Thus, node n to performs a lookup for key k using the most distant finger that does not overshoot the destination. As a results, successive routing hops reduce the distance to the destination by at least half the identifier circle. Furthermore, the finger table allows only one node in the network to meet the criteria for each finger, unlike the routing table entries in Pastry. Therefore, the Chord protocol cannot favour nearby nodes (according to some proximity metric). Chord makes a probabilistic guarantee requiring $O(logN)$ hops for resolving lookups and using $O(logN)$ node routing state. The node join protocol requires $O(log(2N))$ messages.

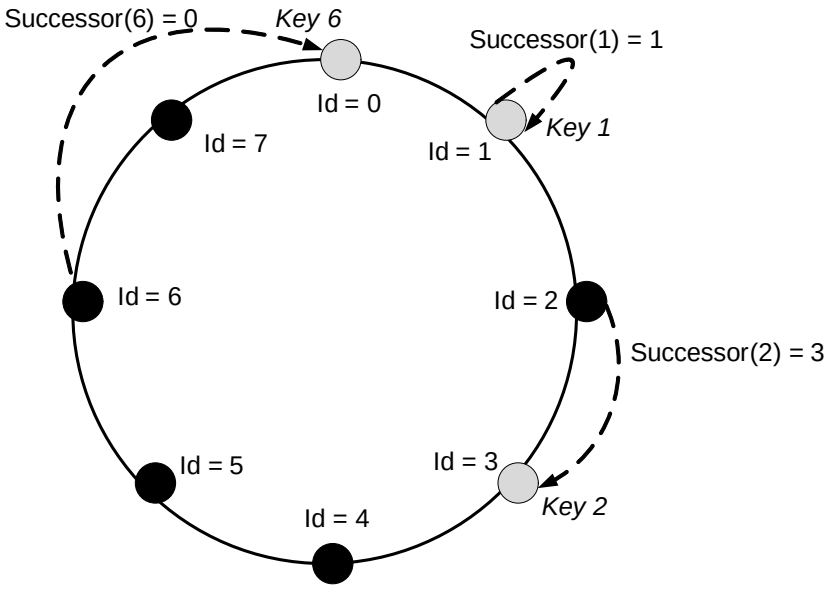


Figure 2.8: Chord circular identifier. Each key is stored in the first successor node in the identifier circle.

Unlike Chord and Pastry, Content Addressable Network (CAN) [RFH+01] organizes nodes into a d-dimensional Cartesian coordinate space on a d-torus. Essentially, this is a d-dimensional space with the range of each dimension wrapping around (like Chord's and Pastry's identifier circle). Each individual node of the CAN network stores only a part (referred as a zone) of the information space, as well as information about a small number of adjacent zones. Each key maps to a point in the space that belongs to a zone and then this key is stored to the node responsible for this zone. The messages are greedily routed towards the neighbour zone that gets the message closer to the destination. The average routing path length is $O((d/4)(N^{1/d}))$. Furthermore, the routing table size in each node is $O(2d)$. Compared to Chord's and Pastry's $O(logN)$ bound for both these metrics, CAN exhibits a better table size, but worse routing path. The constant routing table size is a unique feature of CAN.

Tapestry [ZHS+04] is similar to Pastry in the use of an identifier circle and prefix routing; it also guarantees the same $O(logN)$ routing hops and node state overhead. It adds some additional interfaces to allow storage of duplicate (key, value) pairs across the network for caching purposes. Queries for a key will automatically find the closest copy.

As already described, Pastry nodes are organized in a circular identifier space as in Chord, but use prefix routing, in which the prefix of each hop's identifier matches the destination by at least one extra digit. This allows a choice of nodes for each hop. Also, Pastry considers a locality measure based on the physical network distance between the nodes. This measure is used when deciding the best node for each hop. In a network of

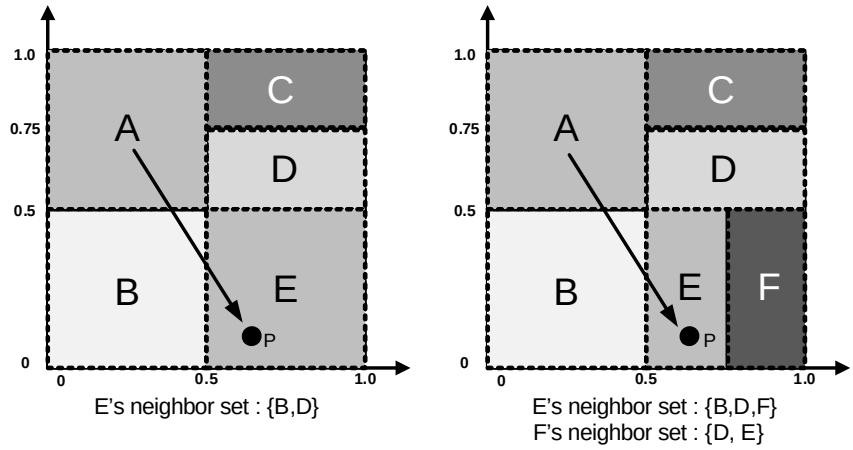


Figure 2.9: CAN 2d coordinate space. Each node stores a rectangle in this information space. Each key maps to a point in the space and is stored to the node responsible for this point.

N overlay nodes, Pastry can theoretically route a message to a destination in $O(log_{2b}N)$ hops, while each node must handle $O(2^b \times log_{2b}N)$ routing state. Clearly, b can be used to trade off between routing hops and state.

MSPastry [CCR04] is a Pastry implementation that includes techniques to achieve high dependability and good performance in a realistic network environment characterized by high node churn rate. It provides dependable routing that makes the routing mechanism both consistent and reliable. Obviously, ensuring route dependability introduces additional overhead, but the techniques proposed for *MSPastry* are self-tunable, adjusting their overhead according to the stability of the network. That is, when the system is fairly stable (i.e. exhibits low churn rates), the proposed mechanism reduces its operation and hence the imposed additional overhead.

A key is mapped to the node whose identifier is closest to the key in the identifier space. This node is called the key's root. The routing is consistent if no overlay node ever delivers a lookup message when it is not the current root node for the message's destination key. Consistent routing is important because inconsistencies can lead to degraded application performance and user experience. *MSPastry* ensures consistent routing by keeping the leaf sets consistent. The key aspects of keeping the leaf set consistent is to: (a) mark a node active (when joining) only after ensuring all the nodes in its leaf set are active by probing them; and (b) maintaining its leaf set by periodic heartbeats. Nevertheless, in order to reduce the overhead of the heartbeat messages, *MSPastry* nodes send heartbeat messages to its immediate left neighbors. When a node does not receive heartbeat message from its right neighbor for a particular amount of time, it triggers a "suspect" routine to determine whether the right neighbor is still in the leaf set or not.

Consistency only, is not sufficient for dependable routing, as messages may be lost

when they are routed through the overlay. So it is important for *MSPastry* to provide reliable routing. This is achieved by using active probing and per-hop ACK messages. Under active probing, a node periodically probes each entry in its routing table to ensure they are still alive. When a route in on the way, the nodes expect to receive an ACK message from the node to which the route is forwarded. If they did not receive ACK, the node will re-route the message using another node from the routing table. The active probing period in *MSPastry* is self-tunable, depending on the observed node failure rates in the network.

2.6 Conclusions

This chapter provided a presentation of the knowledge that is essential in order to understand our work. We presented basic notions on workflow systems, Web services, the publish/subscribe paradigm, and the Peer-to-Peer systems. We discussed different solutions and made arguments about the benefits and the drawbacks of each approach.

Chapter 3

Related Work

This chapter discusses previous proposals that followed similar methodology, mechanisms, and architecture with our solution. First, Section 3.1 presents the related work in content-based publish subscribe systems implemented upon structured P2P overlays. We present diverge solutions that propose different approaches in the used overlay and the subscription and publication model. We especially focus on systems that use the DHT infrastructure. The reason is that our solution extends the systems Ferry and eFerry which use the DHT infrastructure.

Second, Section 3.2 presents the current status in the area of decentralized orchestration systems. We point out the benefits gained from the decentralized orchestration and then we give a short presentation of related decentralized orchestration engines, with special focus on their execution mechanism. Our system is closely related to a previously proposed system called NIÑOS that also used a pub/sub mechanism and for this reason NIÑOS is presented in more detail.

3.1 Content-based Publish/Subscribe over Structured P2P Overlays

Many attempts have been made in designing content-based pub/sub systems over unstructured P2P overlays. Most of these approaches rely on a small number of trusted brokers that are inter-connected using a high-bandwidth network [TKLB07, WG08, GMS06]. In some scenarios, such configurations may or may not offer adequate scalability. Nevertheless, they do not provide a satisfactory level of fault tolerance since crashing of a single broker may result in a large number of state transfer operations during recovery. On the other hand, structured P2P networks have emerged as an infrastructure for building efficient, scalable, and self-organizing distributed systems. In this section we examine related work on modern distributed content-based pub/sub systems build using the hypercube or DHT architecture.

Tam et al.

Tam et al. [TAaJ03] proposed a content-based pub/sub system implemented by extending the topic-based[1] pub/sub system Scribe [RKCD01]. Similar to Scribe, their proposal uses Pastry as the underlying DHT layer. For each subscription the system builds a set of topics for submission to the topic-based system. This process requires that the pub/sub messages provided by the user application (as it is application specific) follow certain rules and constraints according to a predefined schema. Every user application facilitates a different schema, that must be broadcast to all the participating nodes. Note that the system is able to handle simultaneously multiple schemas.

A schema consists of several tables (e.g. Table 3.1 presents a schema that describes computers). A schema table maintains information about a set of attributes, including their type, name, and constraints over a range of possible values. Furthermore, each schema table has a set of indices that are used by the network lookup mechanism. An index is an ordered collection of strategically selected attributes by the schema designer. The selection of proper attributes is critical for the performance of this mechanism; the selection of commonly used attributes may result in the production of increased network traffic due to false positives. For example, if an index uses only an attribute that has two possible values (true or false with 50% chance) then every published publication will have 50% chance to be matched with a subscription, but may latter rejected when the complete match is done.

When a subscription or publication is submitted to the system the following steps take place: a) the pub/sub message is inspected and matched with the available schema tables, b) an associated schema table is found, and c) a number of index digests are produced according to the predefined indices in the schema table. An index digest is a string of characters formed by concatenating the type, name, and value, for each index attribute. For example, if we use the schema presented in Table 3.1 and the first index, we can produce the index digest [Euros:Price:500 : String:CPU:i3 : GHz:Clock:2].

A given subscription can be submitted to the system only if it specifies all the values for at least one of the indices in the corresponding schema table. Afterwards, the composed index digest is translated into a DHT hash key and propagated to the associated node. When a publication with attribute values that match the subscription is submitted to the system the same hash key is generated. Therefore, the publication is propagated to the same network node which makes the match.

Notice that this method provides partial matching, as only a subset of the subscription predicates can be matched this way. Therefore, the original subscriber node is responsible for completing the matching process. Other limitations of this pub/sub method is the missing support for range attribute values and the introduction of a schema designer.

[1] topic-based pub/sub systems are presented in section 2.4

Order	Type	Name	Values	Index 1	Index 2
1	Euros	Price	500...2000	✓	✓
2	String	CPU	i3, i5, i7	✓	✓
3	GHz	Clock	2...3.6	✓	
4	MB	RAM	512...4096		✓
5	GB	HDD	256...1000		
6	Inch	Monitor	17...22		✓
7	String	Graphics	Adequate, Average, Good		✓
8	String	Quality	New, Used, Demo		✓

Table 3.1: An example for a schema table for computers with two indices. The first index is useful to enterprise customers, which are subscribers concerned with price and computational power. The second index is useful to home users, which are subscribers concerned with price, quality, and multimedia performance.

Terpstra et al.

Terpstra et al. [TBF+03] proposed a content-based pub/sub system built on top of Chord [SMLN+03]. Their approach extends content-based filtering strategies that were originally proposed by REBECA [MÖ1]. The notification service in REBECA relies on a set of network brokers that forward the notifications using filter-based routing tables. The brokers exploit two methods to significantly decrease the routing table size:

covering, where the broker tests whether a filter F_1 accepts a superset of notifications of a second filter F_2 and replaces all occurrences of F_2 in the same routing table link by F_1, therefore transmitting only F_1 to the neighbor brokers.

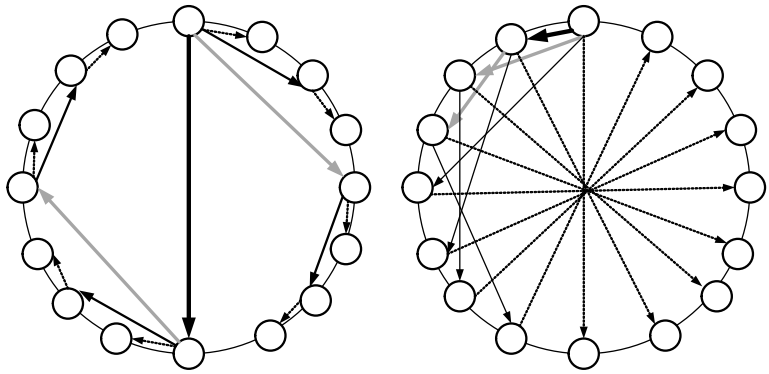
merging, where if no cover can be found in a given set of filters, a new one is created that covers the existing ones and only the merged filter is forwarded to the neighbor brokers.

In REBECA, the network used for data dissemination is comparable to a single spanning tree, therefore exhibits limited fault tolerance as each node is a single point of failure. Additionally, the central nodes are likely to carry most part of the network traffic.

Terpstra et al. proposed the use of a different dissemination tree in every broker. This way each broker acts as the root of its own tree for delivering a matching notification. On the other hand, the use of multiple trees implies that a subscription must simultaneously propagate up to all possible publication trees. Furthermore, if $v \sim u$ is the route taken by a tree routed at v to publish a notification to u, then the subscription tree rooted at u must follow the path in reverse. For example, if the publication algorithm followed the path $(v, u_1, u_2, \ldots, u_n, u)$ to reach the subscriber u from v, the goal of the subscription algorithm is to reach every node in the path $u_1, u_2, \ldots, u_n$ in reverse (in Chord that

is counterclockwise). The publication and subscription propagation is illustrated in Figure 3.1.

As each edge in the path has a filter $f(u \rightarrow v)$, the protocol requires filter invariants to ensure that published notifications follow a path on which the filters are always subsets. We say that a filter accepts an event notification $e \in E$ if $e \in f(u \rightarrow v)$. Otherwise, the notification is rejected. A filter f covers another filter g if $f \supseteq g$. Merging two filters is creating an h such that $h \supseteq f$ and $h \supseteq g$. The covering and merging are used to ensure the filter invariants. In this manner, as notifications follow a path on which the filters are always subsets, no early filter will reject a notification which may have been accepted later. On the other hand, as the system tries maintain the filter invariants, excessive traffic may produced by filter update messages in the case of frequent node joins and departures.



(a) Terpstra et al. publication propagation.

(b) Terpstra et al. subscription propagation.

Figure 3.1: The publication messages 3.1(a) are delivered clockwise on the Chord ring, while the subscription messages 3.1(b) are delivered counterclockwise.

Triantafillou et al.

Triantafillou et al. [TA04] also built their content-based pub/sub system on top of Chord. As shown in Table 3.2, the proposed publication schema consists of a set of attributes, where each attribute consists of a type, a name, and a value. In the case where the attribute value belongs in a range, a minimum and a maximum value are defined along with a precision value.

The subscription schema consists of a set of attributes with name, operation (less, equal, etc.), and value. An example is shown in Table 3.3. Additionally, each subscription has an identifier that consists of three concatenated parts: a) the unique node id of the original subscriber, b) the id of the subscription itself, and c) the number of attributes in the subscription where constraints are defined. Figure 3.2 shows an example of a subscription identifier with a 3-bit identifier space.

Publication					
Type	Min	Max	Precision	*Name*	*Value*
string	-	-	-	exchange	NYSE
string	-	-	-	symbol	aapl
float	0.0	20.0	0.01	price	8.40
float	0.0	20.0	0.01	high	8.80
float	0.0	20.0	0.01	low	8.22

Table 3.2: An example for a publication about the Apple Inc. stock value in NYSE stock market.

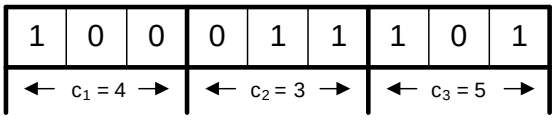


Figure 3.2: Subscription id in a 3-bit identifier address space. Each node can support 8 outstanding subscriptions with an attribute schema including 7 attributes. This example identifies subscription 3 (c2=3), belonging to node 4 (c1=4), comprised of constraints on 5 attributes (c3=5).

Two methods are used for storing a subscription, that depend on the used attribute operation:

- If the subscription attribute has an equality operator ($=, \neq$), then a key is produced by hashing the value of the attribute.
- If the subscription attribute has a range operator ($<, \leq, >, \geq$), then keys are produced by hashing each of the attribute possible value within the defined range.

The above methods are applied for all subscription's attributes. The subscription is later stored using all the produced keys to a number of broker nodes. Notice that in the case of a range operator, the number of the produced keys depends on the preset precision. Thus, the subscription installation and update can be expensive due to the large number of nodes and messages that are potentially involved. This happens when the range is significant and the precision is fine-grained.

After the subscription installation is over the users can submit their publications. To match the publications with the stored subscriptions, the matching algorithm starts by processing every publication's attribute separately. Then for each matched attribute, creates a list and stores matched subscription id as shown below. At the end of this process, all the the subscription ids are stored into a set of subscription-id lists. For example, lets assume that we have the *publication* shown in Table 3.2. After the publication is send to the system two subscriptions are matched; these are $subscription_1$

(presented in the Tables 3.3) and $subscription_2$ (presented in Table 3.4). Thus, the following subscription-id lists are produced:

$L_{exchange} \rightarrow subId_1$

$L_{symbol} \rightarrow subId_1 : subId_2$

$L_{price} \rightarrow subId_1$

$L_{low} \rightarrow subId_2$

As shown, $subscription_1$ was found in three lists, while $subscription_2$ was found in two lists. By processing the attribute names of $subscription_1$ and $subscription_2$ we find that both have constraints over three attributes. So only $subscription_1$ is matched and using the first field stored into the $subscription_1$ we can get the node id of the subscriber node and inform it about the matching event. A major drawback of this protocol is the small domain problem; if the number of nodes in the network is much greater than the domain of attribute values then the network may have a number of useless nodes between two consecutive values (ring positions) of the attribute's range. Also, the number of produced pub/sub messages depends on the precision of the predefined range.

Subscription$_1$		
Name	*Operation*	*Value*
exchange	equals	NYSE
symbol	equals	aapl
price	less	8.70
price	greater	8.30

Table 3.3: An example of a subscription about the Apple Inc. stock in NYSE stock market, requesting current prices between 8.30 and 8.70.

Subscription$_2$		
Name	*Operation*	*Value*
symbol	equals	NYSE
price	equals	8.20
low	less	8.05

Table 3.4: An example of a subscription about stocks in NYSE stock market, with current price 8.20 and low price less than 8.05.

Ferry and eFerry

Zhu et al. proposed a system called Ferry [ZH05], which is based on Chord as well. Ferry is essentially a rendezvous network built on top of Chord to support content-based pub/sub services. Each node can have two roles: a) it can act as a rendezvous point (RP) and provide matching and storage for pub/sub messages, and b) it can act as an intermediate node as part of a delivery path and simply propagate further the received message.

Given a pub/sub scheme $S = \{A_1, A_2, \ldots, A_n\}$, the RP nodes are chosen as the most immediate successors of $k_i = h(A_i)$, where k_i is derived from the attribute A_i using

the Chord's consistent hash function $h()$. A subscription $s = \{A_1, A_2, \ldots, A_n\}$ can be installed using two schemes: a) by randomly choosing an attribute (e.g. A_2) from s and using $k_2 = h(A_2)$ to install it to the appropriate RP node, or b) by choosing an attribute, A_i with value $k(A_i)$ closer to the ids of the neighbors' nodes and then install the subscription to the appropriate RP node.

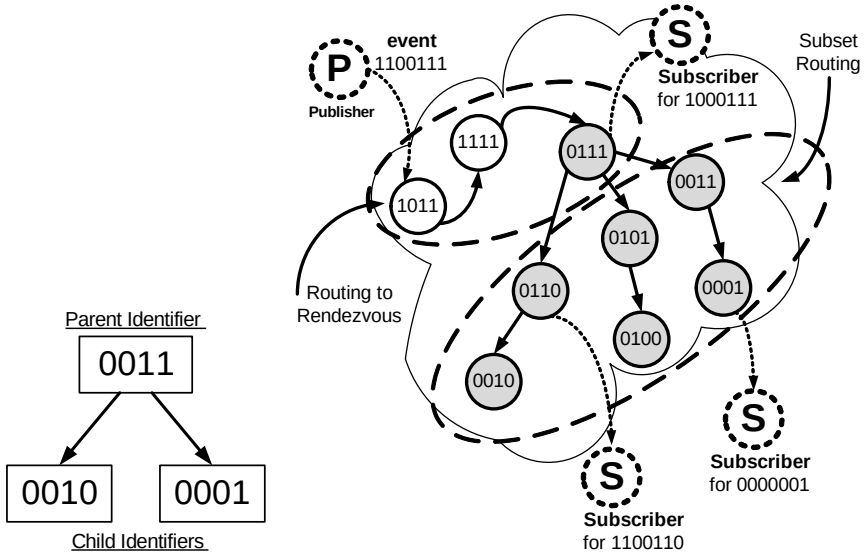
For the publication of an event $e = \{A_1, A_2, \ldots, A_n\}$, the publisher node must produce the hash key $h(A_i)$ for each attribute. Then using these keys the publisher sends the event to every RP node that is the immediate successor of this value. To reduce network traffic, the protocol aggregates into a single message all the events which share common ancestor nodes in the underlying DHT tree, thereby minimizing the number of messages. The main disadvantages of Ferry is the use of $|e|$ messages for each publication and the limited number of used rendezvous point (RP) nodes (only $|S|$). This causes the system to exhibit poor scalability when a large number of subscriptions is combined with a limited number of attributes.

To overcome this problem, an extension to Ferry was proposed, called eFerry [YZH07b]. In this system, instead of hashing each attribute, a vector of attributes is hashed. For example, given the above schema with subscription $s = \{A_1, A_2, \ldots, A_n\}$, only a single key is produced with $k = h(A_1, A_2, \ldots, A_n)$. Since the attributes vector is hashed there, can be $2^n - 1$ RP nodes. This way, the overhead for storing, matching, and delivering subscriptions per RP node is significantly reduced. Furthermore, since only one RP node stores subscriptions with the same attribute set, the subscription management becomes more convenient, as local optimizations and indexing mechanisms can be used to enhance event matching.

Nevertheless, when the pub/sub schema has only a few attributes, the above solution cannot improve the scalability. Moreover, the RP nodes of hot attribute vectors tend to be unduly loaded. For this reason, the authors present more elaborate load balancing mechanisms to address the above problems. The publication overhead is still a problem because it has to investigate each subset of the whole attribute set. Thus, for a event $e = \{A_1, A_2, \ldots, A_n\}$, each possible subset of $(A_1, A_2, \ldots, A_n)$ is taken, and the event is sent using the produced hash value to the immediate successor. This compromises the event matching performance, as the number of the possible subsets in $|S|$ is exponential, i.e. $O(2^n)$.

Reach

Reach [PWR04] is based on a Hypercube overlay. As it is implied by the pub/-sub scheme, it is characterized by n enumerated attributes. Each subscription and publication bears a n-bit identifier, where a 1 in the ith bit indicates interest/association with the ith attribute. The matching is done if for every bit in the subscription, the according bit in the publication is also set. Moreover, the encoding scheme defines an identifier hierarchy, i.e. id_1 is parent of id_2 if and only if id_1 is superset of id_2. This means that a parent identifier contains at least all the attributes of a child identifier, as shown in Figure 3.3(a).



(a) Reach Parent/child identifier space relationships.

(b) Reach event dissemination.

Figure 3.3: Figure 3.3(a) shows the parent/child relationships into the Reach identifier space, while Figure 3.3(b) shows the dissemination of event 1100111 in a network of size 2^4 (m=4, n=8).

The network usually consists of 2^m nodes, where $m \leq n$, and each node is identified by a m-bit identifier. A subscription with identifier sid is mapped to a node with identifier nid, if their lower order m bits coincide. For example, in a network of four nodes with a 4-bit attribute space, the node 01 would host the identifiers 0001, 0101, 1001, and 1101.

Reach uses a Hamming-distance routing scheme, where each node maintains a routing table that contains the addresses of all the nodes whose identifiers are one Hamming distance from its own. These tables are used to incrementally forward a message by sending it at each hop one Hamming distance closer to the destination, thus requiring on average $O(m)$ hops. The matching process is illustrated into Figure 3.3(b). At first, the publication is propagated to its associated node (e.g. for a message with identifier 1100111 this will be node 0111). Then, the publication is progressively forwarded to nodes that host subsets of the event identifier (e.g. nodes 0110, 0011, and 0101). This process is continued until it reaches all subset subscriptions and at each step the publication is matched with the stored subscriptions. The main disadvantages of this proposal are: a) the statically defined attribute set, b) the coarse-grained use of the bit vector to express the user's interests, c) the limited fault-tolerance due to intermediate node failures, and d) the lack of load balancing methods that can handle unevenly distributed workloads.

HOMED

In a similar vein, HOMED [YCP04] maintains a Hypercube overlay where each node has a d-bit identifier derived from its subscriptions using an id generating function. This way, a set of predicates is transformed into a d bit id, with the unique requirement that if a predicate α is covered by another predicate β, then the generated id_β must subsume all 1s of id_α. HOMED facilitates a semantic structure where the nodes with similar interests are neighbors and only interested neighbor nodes participate in disseminating an event.

Every node has an id cover table with d entries and each of them corresponds to a neighbor node id with hamming distance 1. Each publication has an event id generated by the same id function which represents the space that the event should be delivered. So, the routing scheme consists of matching the publication's id space with the ids of the neighbor's nodes. When a new node enters the network, it contacts a well-known node which acts as an entry point and finds the one that covers its id. After it is joined, the new node determines the id space covered by it and updates the cover tables of its neighbor nodes.

Figure 3.4 shows an example of event dissemination in the 3-dimensional HOMED. An event with id $e_{id} = (1 * *)$ occurs at the node 000. The event is routed to the first matching node 100 and then it is multicast. The id spaces on arrows show that the e_{id} is split as the event moves toward the leafs of the multicast tree. In this example, the node 101 is assigned the responsibility for delivering the event to 111, since the node 101 has a better cost metric than 110.

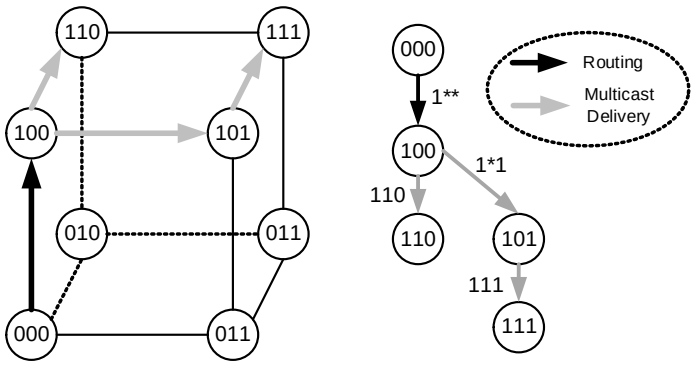
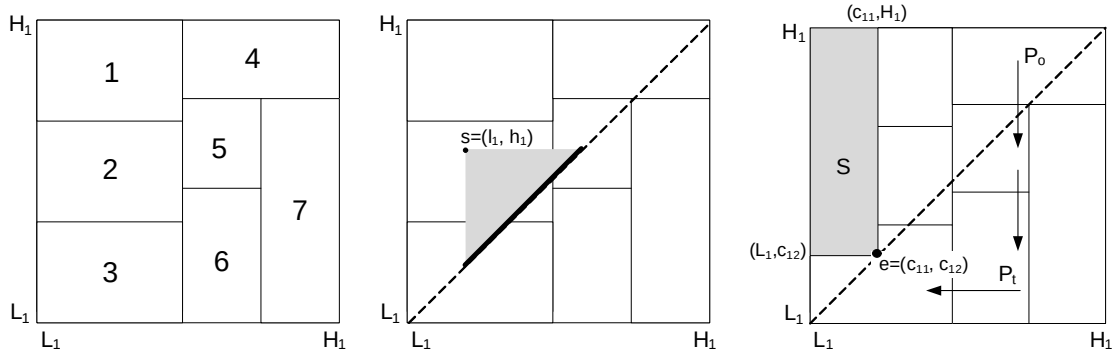


Figure 3.4: An example of event dissemination in a 3-dimension HOMED network.

HOMED has the following limitations: first, it assumes a globally static attribute space. Second, it presents poor load balancing, since non uniformly distributed subscriptions would cause unevenly distributed nodes on the overlay. Finally, similar to Reach, it is difficult to derive node ids from their subscriptions, while preserving the high expressiveness of subscriptions.

Meghdoot

Meghdoot [GSAA04] is based on CAN [RFH+01]. Meghdoot maps a schema $S = \{A_1, A_2, \ldots, A_n\}$ with n attributes into a cartesian space with $2n$ dimensions. Each attribute A_i with domain range $[L_i, H_i]$ corresponds to dimensions of $2i - 1$ and $2i$ of the cartesian space. The predicates of a subscription specify ranges of interest over the attributes, while the ranges are represented by areas in the logical space. This logical space is partitioned among the peers present in the system and each peer is responsible for one of the partitions. The partitions are referred as zones and when a peer is responsible for a partition it owns the zone. For example, if the schema has only one attribute with $[L_1, H_1]$ bounds then we can have the partition shown in figure 3.5(a). In this case, each rectangle is a zone owned by a peer.



(a) Meghdoot schema with only one attribute.

(b) Meghdoot region of events affecting a subscription $s =< l_1, h_1 >$.

(c) Meghdoot region of subscriptions affected by an event $e =< c_{11}, c_{12} >$.

Figure 3.5: Figure 3.5(a) shows the logical cartesian space for the case where the schema has only one attribute. The rectangular regions form a partitioning of the space. Figures 3.5(b) and 3.5(c) show subscription and event propagation in 2d cartesian space for a single attribute schema.

A subscription S can be expressed in the following format: $s = \{(l_1 \leq A_1 \leq h_1), (l_2 \leq A_2 \leq h_2), \ldots, (l_n \leq A_n \leq h_n)\}$, with $l_i \leq L_i$ and $h_i \leq H_i$. The subscription s is mapped to a subscription point in the $2n$-dimensional space $< l_1, h_1, l_2, h_2, \ldots, l_n, h_n >$. Also notice that all subscriptions are stored in the upper left side of the diagonal hyperplane as $\forall i \in \{1, 2, \ldots, n\}, l_i \leq h_1$. Thus, the user submits the subscription to an origin peer (P_o) that maps the subscription to the corresponding subscription point. The peer whose zone contains the point is referred as the target peer (P_t). In order to route the subscription from P_o, each time we propagate the subscription to the neighbor node with the closest Euclidean distance to the subscription point. When P_t receives the subscription, it stores the subscription along with an identifier (e.g. IP address, user name, e-mail, etc.).

Similarly, an event $e = \{A_1 = c_1, A_2 = c_2, \ldots, A_n = c_n\}$ is mapped to an event point

$< c_{11}, c_{12}, c_{21}, c_{22}, \ldots, c_{n1}, c_{n2} >$. When an event is introduced to the system, the origin peer P_o maps the event to the corresponding event point and routes the event to the corresponding peer P_t, which owns the event point. Then, the event is checked for matching with a subscription. A subscription s is affected by the event e if the following property holds: $\forall i \in \{1, 2, \ldots, n\}, l_i \leq c_{i1} \wedge c_{i2} \leq h_i$. Afterwards, the event is propagated from P_t to all peers which own a region affected by the event.

For example, the shaded area in Figure 3.5(b) shows the region of event points in a 2d cartesian space corresponding to a single attribute schema. That can affect a subscription $s =< l_1, h_1 >$, because all the event points in the shaded region will satisfy the above property. The shaded region in Figure 3.5(c) shows the region of affected subscriptions in the 2d space by an event $e =< c_{11}, c_{12} >$.

Considering the skewed distributions of both subscriptions and events in a real application, Meghdoot addresses the load balancing issue by zone splitting and zone replication. The major limitation of Meghdoot is that the overlay structure is determined by the pub/sub scheme and the overlay dimensionality is proportional to the number of event attributes. More importantly, Meghdoot is not able to support multiple schemas with different dimensions. It also uses the lower part of the hyperplane only for routing purposes.

HyperSub

HyperSub [YZH07a] leverages a multi-dimensional locality preserving hashing mechanism. This way Hypersub can partition a multi-dimensional content space into layered content zones which are mapped into system nodes. This mechanism maps nearby data points in the content space to one node or nodes close together in the overlay network. This makes subscription and publication more efficient, but sacrifices DHT's load balance nature. The content space is transformed into a β-ary (β is the base of the key/node identifiers) tree with maximum height $m + 1$ (m is the number of digits in the key/node identifiers), where each node represents a content zone. A subscription s is mapped to the smallest content zone which can completely cover the range specified in s, while a publication p is mapped to an m-level content zone which holds the corresponding point. In this system, the number of nodes in which a subscription is installed to, could be of an exponential magnitude. Nevertheless, as Meghdoot, it provides no flexibility to schema changes, such as adding or deleting attributes.

3.2 Decentralized Service Orchestration

Typically, a composite Web service specification is executed by a single coordination node. It receives the clients' requests, makes data transformations, and invokes the component Web services according to the provided specification. This model of centralized orchestration has several limitations, as the coordination node commonly becomes a performance bottleneck.

On the other hand, in a decentralized orchestration, there are multiple engines. Each one of them executes a portion of the original composite Web service specification in a distributed fashion. These engines communicate directly with each other while they transfer data and define the control flow in a loosely coupled manner. This model brings several benefits as: a) there is no centralized coordinator to become a potential bottleneck, b) data distribution reduces network traffic and improves transfer time, c) control distribution improves concurrency, and d) asynchronous messaging between the engines improves the throughput.

The first step for distributed orchestration is the automatic parallelization of the BPEL process. There are many proposals [CCMN04, YG07, BMM05] that promise to solve this problem. Most of them focus on the use of program dependence graphs (PDGs) [OO84] for partitioning the BPEL process into concurrent composite Web service applications. This problem has some unique features compared to other partitioning problems, such as the presence of fixed tasks and portable tasks. The step followed after the partitioning is the dissemination of the tasks into network nodes and their synchronized execution, according to the defined data and control dependencies.

As the partitioning problem is not the primary focus of our thesis, in this section we present previous work that focus more on the execution mechanisms of the distributed orchestrations engines starting with earlier work on distributed workflow management.

Buhler et al.

Buhler et al. [BV04] described a distributed, agent-based workflow enactment mechanism utilizing BPEL as the specification of the multi-agent system. They exploit a hybrid coordination model, which combines a separate data-centered and control-centered coordination mechanism. The data-centered mechanism leverages a network addressable XML repository that stores XML documents in logical groups and provides data coming from the evaluation of XPath queries. The controlled-centered coordination mechanism provides workflow control based on colored Petri nets [Jen87]. It treats BPEL as a description of the order of a collection of agents, where the agents act as proactive proxies for the underlying Web services. Nevertheless, their work does not support the BPEL structured activities switch and pick and they do not provide a quantitative evaluation of the system. Furthermore, their main focus is the integration of Web services with the workflow management system and not the decentralized orchestration of Web services.

Guo et al.

Guo et al. [GRCB05] introduced a lightweight protocol for agent oriented Web services coordination. The proposed protocol is based on an imperative language for representing the dependencies among distributed agents called Lightweight Coordination Calculus (LCC). In this language we can define the initial roles of the agents, the roles that get after receiving or sending a message, and the constraints under which a message is allowed to be send or received by an agent. Nevertheless, the task of performing the language

mapping from BPEL to LCC is actually the task of translating a imperative program to a declarative program, which is not possible in all circumstances. As a result, the authors propose a limited version of the LCC protocol which acts mainly as a BPEL interpreter. This way, the BPEL specification can be interpreted by the LCC protocol and then both are passed into the multi-agent system. Using this protocol, the BPEL activities can be interpreted and executed on the distributed multi-agent platform, while all the needed messages are packed and passed together among the agents. In this approach, each agent acts as a Web service proxy and the messages are passed directly between agents to control flow. Although their approach is a possible solution for distributed workflow management, they are still working on how to execute the full version of the protocol and how to invoke the Web services. Furthermore, the translation of BPEL specifications to LLC protocol is not verified and there exist no evaluation of their work.

RainMan and Arjuna

RainMan [PPC97] is a service model which separates the responsibility of workflow coordination from activity execution, by creating two classes of entities called Sources and Performers. In effect, while the coordination of each process remains localized within a Source object, the actual execution of activities is decentralized across a network of Performers over which Sources have very limited control. This model is essentially a centralized orchestration of Web services, because Web services act as the Performers and the workflow interactions are performed by the Source. In the Arjuna project [RSW97], the proposed execution model is designed and implemented as a CORBA-based service [Vin97]. It decentralizes the coordination of a process by installing task controller objects in different domains that coordinate with each other to deliver workflow routing functionality. This execution model eliminates a central point of failure, but it imposes computational burdens on participant domains. This fact degrades rapidly the performance, as the number of participants increases.

Nanda et al.

Nanda et al. [NCS04] proposed a decentralized orchestration of composite Web services, with the use of multiple engines. Each one of them executes a composite Web service specification that is a portion of the original composite Web service. The goal of the system is to minimize communication costs and maximize the throughput of the multiple concurrent instances of a given composite Web service. Contrary to earlier approaches that tried to minimize the completion time of a single instance program running in isolation, they focus on running multiple instances.

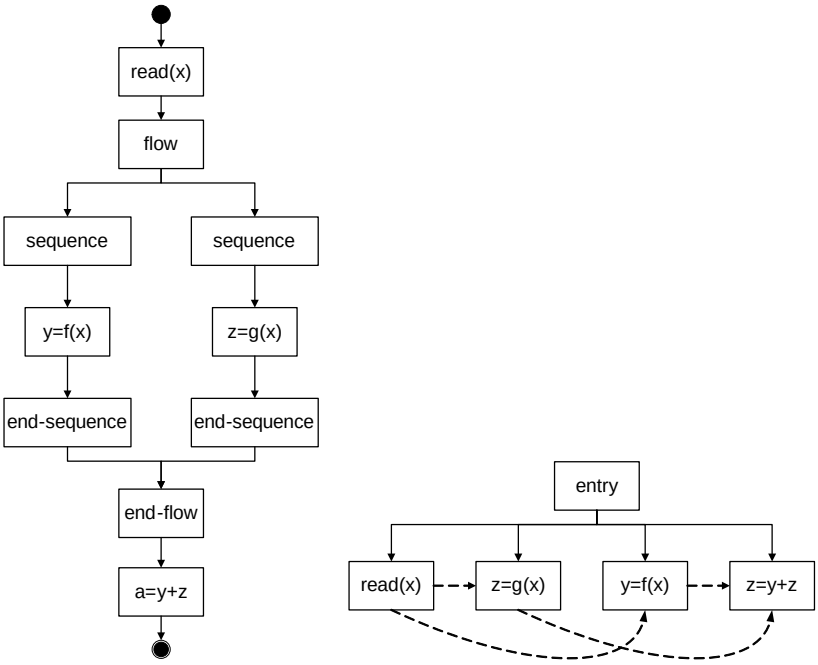
Initially, a code partitioning algorithm identifies the number of final partitions based on the number of component Web services in the composite Web service. To do so the algorithm requires:

1. automatic parallelization and code partitioning using data flow analysis to deter-

mine the most cost-efficient partition (as shown in Figure 3.6 the BPEL control flow graph is transformed into a program dependence graph).

2. synchronization analysis to determine the best synchronization protocols

3. code generator.

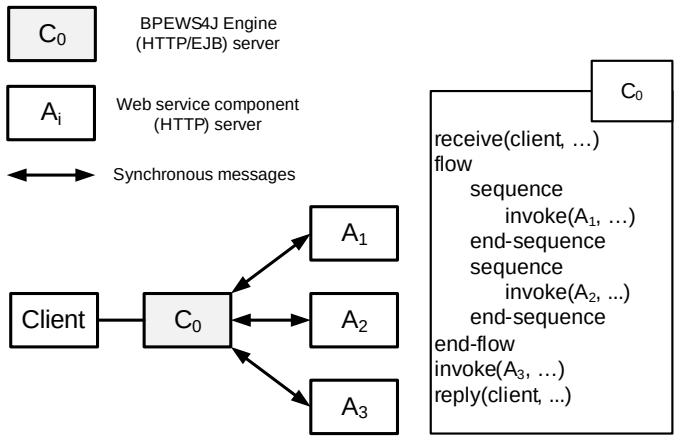


(a) Nanda Control Flow Graph. (b) Nanda Program Dependence Graph.

Figure 3.6: Figure 3.6(a) shows the Control Flow Graph (CFG). This is transformed by Nanda into a Program Dependence Graph (PDG) as shown in Figure 3.8(b). The latter is used to partitioned the BPEL process code.

These partitions are assigned to multiple communicating engines and require that every participating node has a BPEL engine, while the communication appears as Web service invocation in each partition. For example, Figure 3.7 presents a centralized orchestration engine and the BPEL process that will be executed, while the decentralized Nanda architecture is shown Figure 3.8. As presented in Nanda, each node is responsible for executing a portion of the the original BPEL process, while every agent is extended with BPEL orchestration abilities.

The requirement for a BPEL engine on each node happens for two reasons. First, the ability of executing BPEL has become standard software infrastructure in application servers. Second, the application that the server exports as a web service may itself be implemented as a BPEL program behind the scenes, requiring a BPEL execution environment. The authors propose two execution models based on a HTTP or JMS application server.



(a) Nanda centralized architecture. (b) Nanda centralized code.

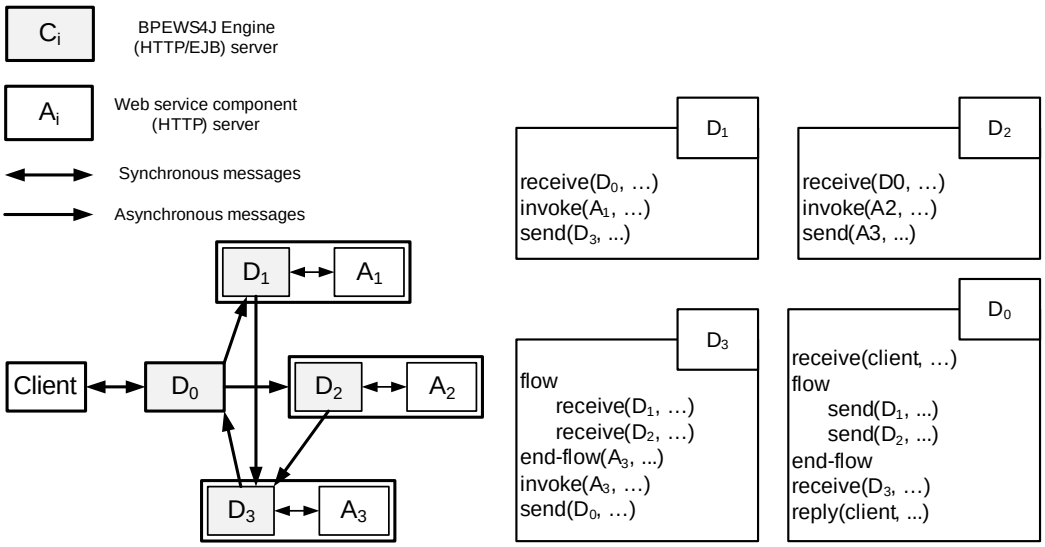
Figure 3.7: Figure 3.7(a) shows the centralized Nanda architecture, while Figure 3.7(b) shows the BPEL code.

These two solutions are tested and display significant increase in throughput in contrast to centralized approaches, with 30% increase under normal system loads and by a factor of two under high system loads. Nevertheless, the process decomposition appears to be very crucial for the overall engine performance, while the requirement for each node to run its own instance of a BPEL engine, clearly make the nodes' administration and fault tolerance mechanisms more complex.

ZenFlow

Ricardo Jimenez-Peris et al. [JPPnMMJ08] proposed ZenFlow, a reflective BPEL web service orchestration engine. Its reflective capabilities enable to implement all non-functional aspects in a separated manner, thus reducing the complexity and increasing the maintainability and modularity of the BPEL engine, as shown in Figure 3.9(a).

The system is structured into two layers: a) the base-level that executes the application components, and b) the meta-level that runs the components related to the implementation of non-functional requirements, such as decentralized execution, security, fault-tolerance, etc. The base-level provides an image, called meta-model, of the structural and behavioral features to the meta-level. Change in one of the levels leads to change to the other level. In ZenFlow three types of metaobjects are used: a) metaobjects associated to all the activities of a business process, b) metaobjects associated to a certain type of activity of a business process and invoked when that type of activity is interpreted, and c) metaobjects associated with a single activity and invoked when the selected activity is interpreted. The distributed interaction of the metaobjects across the BPEL servers is enabled through Remote Method Invocation (RMI) calls using a set of meta-interfaces.

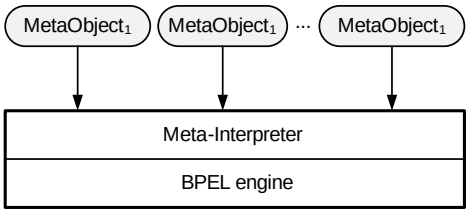


(a) Nanda decentralized architecture. (b) Nanda decentralized code.

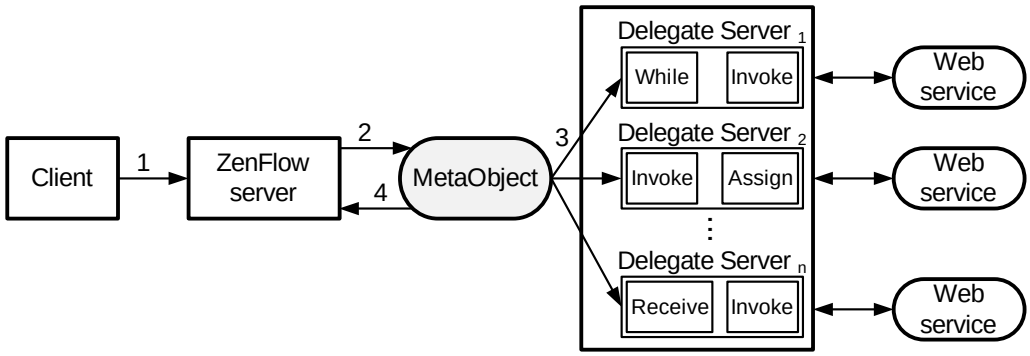
Figure 3.8: Figure 3.8(a) shows the decentralized Nanda architecture, while Figure 3.8(b) shows the partitioned BPEL code.

Figure 3.9(b) presents a more detailed view of the execution steps in the ZenFlow architecture. The client requests the execution of a BPEL process (step 1) and if the process is configured to executed at a different site, the execution is delegated to a server that must support both the interpretation of the business process and the decentralized execution. Using the reflective approach, the metaobjects responsible for delegating the execution to a remote site intercept this event (step 2). At the metalevel, the metaobject extracts the state of the business process, and sends it to the delegated server through the remote RMI meta-interface (step 3). In this case, the delegated server receives the state of the business process and recreates the process locally. Then, the server resumes the execution of the recreated process. When the execution finishes, the delegated server extracts once again the process state which is returned to the metaobject. The metaobject in turn installs the new state into the server to reflect the result of the remote execution (step 4).

The cost of reflection is only paid when metaobjects are associated to activities, so that the overhead is minimal. The exceptions by the receive, reply, invoke, and link activities are handled by the associated metaobject. On the other hand, as the cost of the reflection depends on the produced events during the process execution, the engine's performance can be greatly degraded in special occasions. Furthermore, the single ZenFlow engine instance still remains a scalability bottleneck and a fault-point.



(a) ZenFlow high level architecture.



(b) ZenFlow detailed architecture.

Figure 3.9: Figure 3.9(a) shows a high level view of the ZenFlow architecture, while Figure 3.9(b) presents a more detailed view with the steps during a deployment and execution of a business process.

NIÑOS

A more recent proposal is NIÑOS [LMJ10]. NIÑOS is a distributed agent-based orchestration engine in which several lightweight agents execute a portion of the original business process and collaborate in order to execute the complete process. NIÑOS exploits the capabilities offered by the PADRES [LJ05] distributed content-based pub/sub routing infrastructure. All communications occur as pub/sub interactions including: data transfer, control, monitoring, and coordination among the agents. As the agents need only to be aware of other nodes' content-based addresses, they become location-independent and their reconfiguration and administration becomes much easier.

As shown in Figure 3.11(b), NIÑOS system architecture consists of four components:

1. a network of broker nodes,
2. a number of activity agents; that are nodes capable of executing BPEL activities,
3. Web service agents; that are nodes capable of communicating with external Web services, and
4. a business process manager

The network overlay consists of PADRES brokers (their internal architecture is shown in Figure 3.10), where clients connect using Java RMI or Java Messaging Service (JMS).

Furthermore, each broker has a rule based engine that performs the matching of the incoming publications with the stored subscriptions and decides the next-hop destination of the messages. On the other hand, each activity agent corresponds to a BPEL activity and act as a light-weight pub/sub client. An activity agent waits for its predecessor activities to complete by subscribing to events and then executes its own activity. When the execution completes, it triggers the successor activities. Each Web service agent acts as a proxy by translating between Web service protocols and the pub/sub message formats. This way, Web service agents allow activity agents to invoke and be invoked by external Web services. Finally, the business process manager is a pub/sub client that transforms the business process described in BPEL into pub/sub messages, deploys the process to the available activity agents, triggers the process instances, and monitors the execution.

Evaluations of the proposed architecture showed an improvement of 70% in process execution time and 120% in throughput versus the centralized approach. Nevertheless, the single BPEL manager can become a scalability bottleneck, as it limits the engine and fails to deploy in parallel multiple processes. Furthermore, each agent is specialized to execute a single activity that can be limited for the resource utilization. Finally, the PADRES network is unstructured and exhibits limited fault tolerance, while it fails to handle high node churn rates.

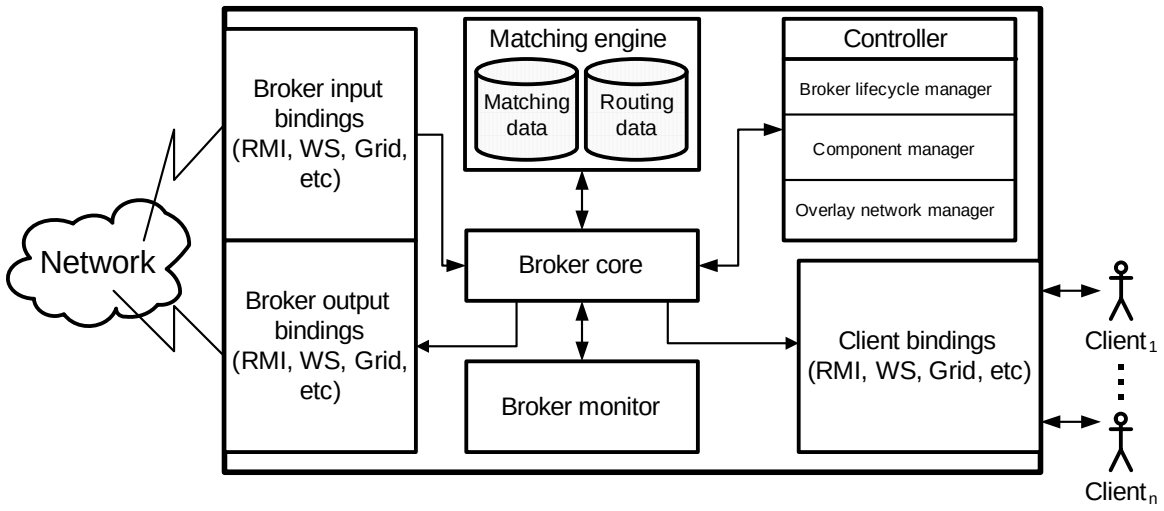
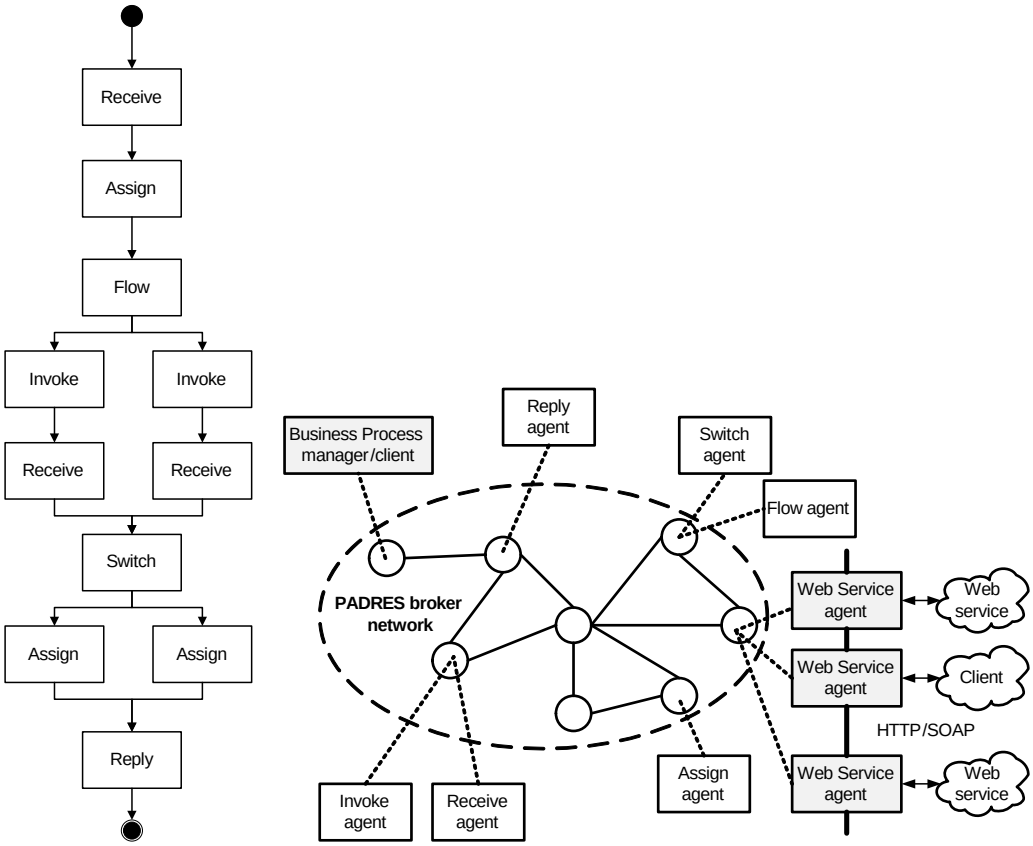


Figure 3.10: The architecture of a PADRES broker. The matching engine maintains various data structures where subscriptions are mapped to rules and publications are mapped to facts. The rule engine performs the matching and decides the next hop destination of the message.



(a) NIÑOS BPEL process.

(b) NIÑOS architecture.

Figure 3.11: Figure 3.11(a) shows a BPEL process and its execution by the NIÑOS architecture presented in Figure 3.11(b). The PADRES broker network carry out content-based routing and in-network processing of composite subscriptions. The agents provide event-driver activity execution and cross-enterprise interaction with external business processes, and transform the presented BPEL process into a set of pub/sub messages.

3.3 Conclusions

There are still a number of research papers on various issues in distributed workflow management. Nevertheless, to the best of our knowledge, no prior work on Web service composition and distributed workflow management has exploited the content-based pub/sub paradigm based on structured DHT networks to achieve decentralized BPEL orchestration. The focus of our work is on mapping BPEL into a pub/sub subscription language and to describe the decentralized execution of the business processes. Our conclusions on the available content-based structured pub/sub systems are presented in Table 3.5. The main reasons that we decided to follow a similar approach with Ferry are:

1. Ferry does not place restrictions on subscriptions, thus it does not sacrifices the expressiveness of the subscriptions.

2. The subscription installation and management algorithms aggregate the event delivery messages, thus minimizing the number of messages across the system.

3. The event delivery algorithm makes use of the embedded trees in the underlying DHT network, thus it is virtually maintenance free.

4. Ferry exploits the fault-tolerance and self-organizing nature of the DHT links and is resilient to node failures.

We extended the Ferry mechanisms and make a couple of contributions that differentiate our approach:

- Our pub/sub algorithm extends with locality meters the event delivery mechanisms in order to minimize required hops from the matching done in the broker nodes to the delivery of the message to the subscriber nodes.

- We extend the matching process with filter merging and covering techniques, thus we reduce the size of the routing tables.

- We implement our solution on top of *MSPastry*, as it ensures consistent routing and it is more efficient in handling node departures under high churn rates.

Furthermore, in Table 3.6 we present an overview of the studied distributed service orchestration engines. From all the proposed solutions we decided to follow the approach taken by NIÑOS as:

1. Its distributed architecture is congruent with an inherently distributed enterprise where business processes are geographically dispersed and coordinating partners have to communicate across administrative domains.

2. It removes the scalability bottleneck of a centralized orchestration engine and allows portions of the process to be executed close to the data they operate on, thereby conserving data and control traffic.

3. It supports flexible mappings of the orchestration agents onto heterogeneous platforms and resources and can be adapted from a centralized to a fully distributed system.

4. The agents only need to be aware of one another's content-based addresses. This simplifies the agent reconfiguration and movement, thus easing the administration burden.

5. The processes are transformed such that certain computations are carried out in the pub/sub layer. This simplifies the orchestration agents control and data flow interactions.

Nevertheless, many problems are not handled by NIÑOS, including fault-tolerance, the required network administration, the centralized BPEL process deployer, and the network transient nature. Our solution differs because:

- We propose a fully distributed architecture that removes the scalability bottlenecks of a centralized deployer agent. Our engine can use multiple deployer/manager nodes that can deploy and trigger business processes concurrently.
- The agents are general-purpose and can execute multiple activities.
- Our engine allows portions of processes to be executed close to the data they operated on, using network proximity metrics. It also stores the pub/sub messages closer to the original subscribers. This is achieved dynamically using an infrastructureless approach and requires no human intervention. Additionally, it exploits the DHT layer proximity properties in order to provide proximity-aware subscription installation and event propagation, thus yielding good message delivery performance.
- It facilitates dynamic mapping of process activities to agents, based on the network conditions and the agents utilization state.
- Our infrastructureless approach requires no custom network configuration by a network administrator. It is designed to operate providing high fault tolerance and can cope with high node churn rates.

Structured Content-based Pub/Sub Systems			
System	**Architecture**	**Pub/Sub Routing**	**Limitations**
Tam et al. [TAaJ03]	Pastry	Based on the index digest values.	a) no support for range attributes, b) requires schema designer, and c) may produce excessive traffic due to false positive messages.
Terpstra et al. [TBF+03]	Chord	Based on filter routing tables using multiple trees.	a) need to maintain filter invariants and b) overhead in high churn rate.
Triantafillou et al. [TA04]	Chord	Based on the matched subscription-id-lists.	a) small domain problem, b) depends on the attributes precision, and c) subscription installation and updates may require large number of messages.
Ferry[ZH05] and eFerry[YZH07b]	Chord	routing leverages the underlying DHT trees.	a) poor load-balancing and b) scalability problems with a large number of subscriptions.
Reach [PWR04]	Hypercube	Based on the hamming distance between the pub/sub id and the ids of the neighbor nodes.	a) the statically defined attribute set, b) the coarse grained expressiveness, c) the limited fault tolerance, and d) the lack of load balancing methods.
HOMED [YCP04]	Hypercube	Based on the pub/-sub id covering by the ids of the neighbor nodes.	a) requires a globally static attribute space, b) presents poor load balancing, and c) subscriptions are unevenly distributed among overlay nodes.
Meghdoot [GSAA04]	CAN	Based on the pub/-sub attribute values coordination.	a) overlay structure is determined by the pub/sub scheme, b) overlay dimensionality is proportional to the number of event attributes, and c) can not support multiple schemas with different dimensions.
HyperSub [YZH07a]	CAN	Based on the pub/-sub attribute zone covering.	a) provides no flexibility to schema changes.

Table 3.5: Content-based pub/sub systems comparison.

Distributed Service Orchestration Engines		
System	**Architecture**	**Limitations**
Buhler et al. [BV04]	Distributed agent engine with control flow coordination based on Petri nets.	a) does not support all BPEL activities and b) the mechanism is described only in theory/no implementation.
Guo et al. [GRCB05]	Distributed agent engine based on coordination using the LCC language. The control flow is defined by the exchanged messages.	a) described only in theory/no implementation and b) a real implementation will be limited by the LCC language.
RainMan [PPC97]	Provides centralized orchestration and flow control using source objects. Provides distributed service execution using perfomers.	a) the centralized module responsible for the orchestration becomes scalability bottleneck.
Arjuna project [RSW97]	Distributes execution in different domains that coordinate execution. Each service is executed by a task controller	a) exhibits low scalability.
Nanda et al. [NCS04]	Uses multiple engines to delegate portions of the process, while tries to minimize the communication cost among them. Requires that every agent participating in the distributed engine must have a BPEL engine.	a) the process decomposition is crucial for the performance of the engine and b) each node requires a running BPEL engine.
ZenFlow [JPPnMMJ08]	Reflective engine that is responsible for the non-functional aspects of the execution. Uses delegate servers for the activities execution.	a) the cost of the reflection depends on the events during the process execution and b) the ZenFlow engine still remains a scalability bottleneck.
NIÑOS[LMJ10]	Distributed architecture based on an overlay of PADRES brokers that are responsible for a number of task agents. Each task agents executes a BPEL activity.	a) single BPEL manager that may become a scalability bottleneck, b) each agent is specialized to execute a single activity, and c) the PADRES network is unstructured with limited fault tolerance and fails to handle high churn rates.

Table 3.6: Distributed orchestration engines comparison.

Chapter 4

Design and Architecture

This chapter presents the design and architecture of the proposed orchestration engine. First, in section 4.1 we give an overview of the architecture and its constituting elements. This section provides enough information for the reader to understand the basic components of our system, how they operate, and how they interact with each other. Second, in section 4.2 we introduce the pub/sub model that is employed by our solution and we discuss the proposed subscription and publication algorithms. Third, in section 4.3 we give a short presentation of the proposed mapping between the BPEL language constructs and the facilitated pub/sub model. Finally, in section 4.4 we conclude with a detailed description of the engine's operation during the process deployment, the process instance execution, the process redeployment, and the process undeployment phase.

4.1 System Overview

We propose ADORE (Adaptive Distributed Orchestration over infRastructureless nEtworks), a distributed business process execution architecture. Our system leverages an underlying content-based pub/sub infrastructure, by transforming BPEL business processes into a set of fine-grained pub/sub messages, that determine the data and control flow of the business processes. Utilizing these pub/sub messages, the collaborating overlay nodes realize the original processes by taking advantage of some of the in-network processing capabilities. We pursue to achieve maximum scalability and parallelization and for this reason we decided to include multiple nodes with the ability to deploy and to execute BPEL processes. Nevertheless, as the process management is critical, our architecture can also provide process deployment and monitor in a centralized manner, again exploiting some of the decoupling properties of the pub/sub system.

The ADORE architecture, as presented in Figure 4.1, consists of three major components:

- *An underlying content-based* pub/sub *system.* The latter is build over a DHT network overlay. We selected the *MSPastry* DHT overlay, thus the network nodes form a ring.

- *A number of deployer nodes*. These are DHT nodes capable of sending and receiving pub/sub messages and of deploying BPEL processes. Based on the system configuration, we can have a single or multiple deployer nodes. Hereinafter, we assume that we have a single deployer node, unless we clearly state the opposite.
- *A number of worker nodes*. These are DHT nodes capable of sending and receiving pub/sub messages and of executing BPEL activities.

Additionally, we present three types of external actors that interact with our architecture. These are:

- *Web service providers*. They are invoked by the worker nodes and provide service operations.
- *Web service requesters*. They invoke worker nodes and request service operations.
- *External clients*. They interact with the deployer nodes and may request the deployment of a BPEL process, or provide initial input to the BPEL process instance. Furthermore, they may request the produced output after the process instance termination.

In the following sections we give a short overview of the system's basic components and then we proceed by describing the details of their internal architecture.

Content-based Pub/Sub - DHT Overlay

The DHT network overlay acts as the backbone for the pub/sub message routing. We must note here that our architecture is independent of the deployed DHT network and thus, it can rely on any of the available DHT implementations[1]. Nevertheless, among the available solutions, we decided to use the *MSPastry* overlay [CCR04]. *MSPastry* is based on Pastry DHT [RD01b], but extends it in order to provide better performance in high node failure and churn rates. Our choice of *MSPastry* was also driven by its advantages:

1. *Guaranteed lookup*. It provides a guarantee that a search request will need no more than $logN$ hops, where N is the number of nodes in the network. No message flooding is required and the messages reach their destination at a guaranteed number of hops. This way, we do not need to implement complex search algorithms or use advertisement messages.
2. *Efficient routing table restoration*. It provides the best performance (in terms of message overhead) for fixing the broken DHT routing tables during periods with high node churn rates. It facilitates low overhead for failure detection using three methods: a) by exploiting the overlay structure, b) by using self-tuning probing periods, and c) by using any messages exchanged between two nodes in place of failure detection messages.

[1] As those presented in Section 2.5.3

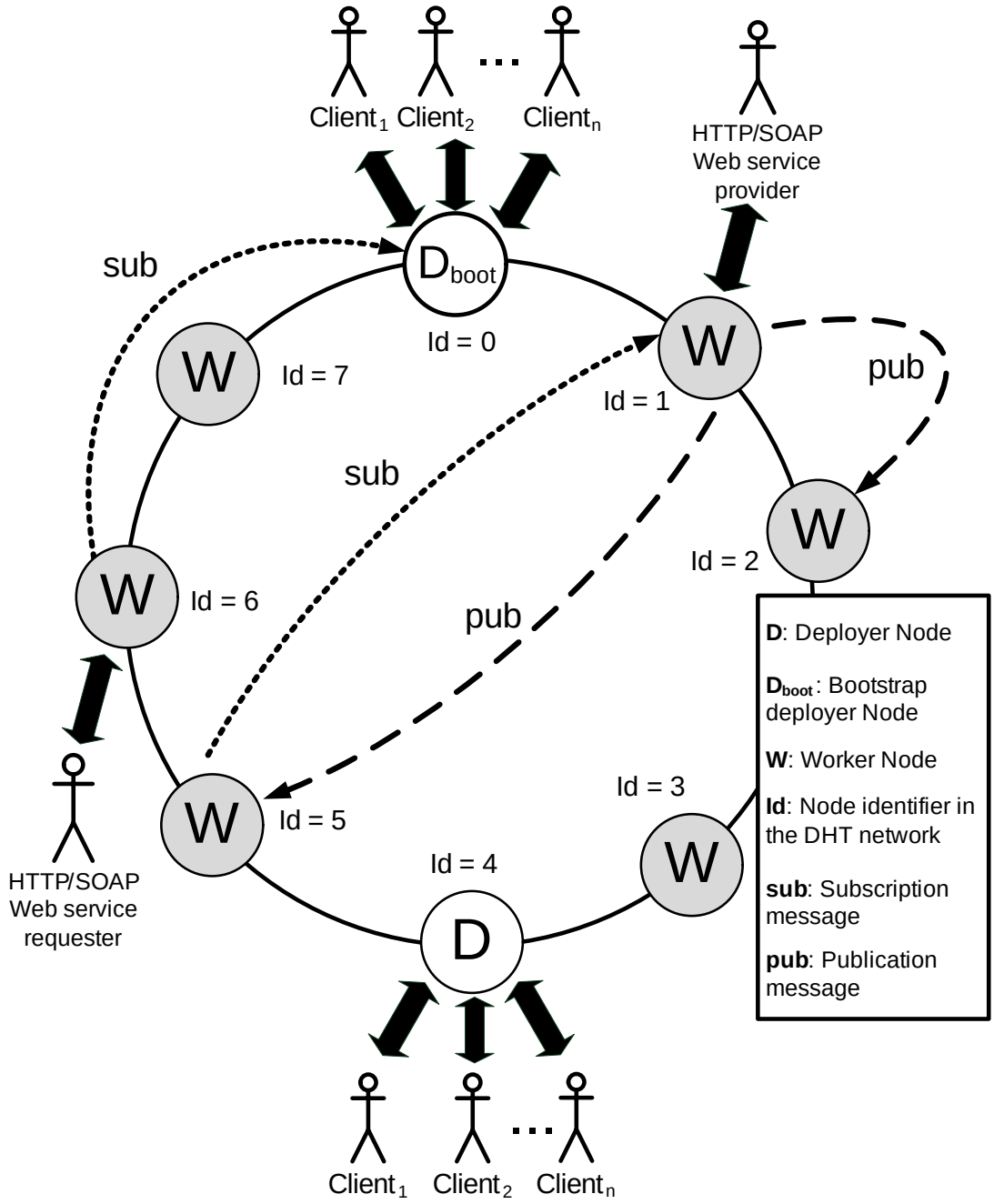


Figure 4.1: Architecture of the distributed orchestration engine. All nodes have unique id's and participate in a DHT network, where they exchange pub/sub messages. There are two kinds of nodes: workers and deployers. The deployers receive requests from external clients and deploy, undeploy, and trigger the execution of BPEL processes. The worker nodes execute assigned BPEL activities and publish pub/sub messages to trigger the execution of the following activities. Moreover, they communicate with external clients when they execute a <receive>, <invoke>, or <reply> activity.

3. *Proximity propagation*. It facilitates locality meters for minimizing the average hops that a message needs to reach its destination.

4. *Good scalability*. The search/routing algorithm scales well even with a very high number of nodes, in contrast with unstructured P2P networks.

5. *Generic interface*. There are numerous open and mature implementations of the Pastry DHT (e.g. freePastry) that can be extended to support the *MSPastry* protocol. Our implementation is based on a minimum interface provided by the *MSPastry* protocol. So, we need to modify only a small part of our implementation in order to use an alternative implementation like freePastry.

Going back to the description of the DHT overlay, each node has a unique id and a routing table with the addresses of a number of neighbor nodes that have local-proximity. The DHT layer provides only the simple routing operations shown in Figure 2.3. For example, the method $route(key, msg)$ stores the given message to the network node with node id numerically closer to the key.

We leverage these routing services by building a higher level routing mechanism based on the pub/sub content-based model. This scheme provides loosely-coupled interactions without the usage of specific node addresses and supports subscriptions with high expressiveness. Thus, at the application level we only use publish and subscribe primitives, without providing any network address information. By facilitating this mechanism, all distributed interactions take place in the following manner: a) at any time, a subscriber expresses its interest by submitting a subscription for specific content, b) a number of nodes produce publications that are sent throughout the network, and c) the matching publications are disseminated only towards the subscribers that have previously expressed their interest in receiving similar content.

To conclude, the DHT network consists of nodes with unique network ids which have the ability to store, match, and propagate pub/sub messages. These nodes have three main roles: a) they can act as network brokers storing and matching subscriptions, b) they can act as router nodes by carrying out content-based routing, and c) they can act as information producers and consumers by sending pub/sub messages to the network.

Deployer and Worker Nodes

The main role of a network node is to act as an entity that can support the deployment or the execution of BPEL activities in a distributed manner. A network node is classified as a deployer or as a worker node. This classification results from the node's role during the engine's operation.

In a nutshell, the deployer node: a) transforms the business process into a set of pub/sub messages, b) deploys the process activities into the available worker nodes, c) triggers an instance of the deployed process, and d) monitors the process instance execution. At first, a request for a BPEL process deployment is received from an external client. Then, the deployer parses the BPEL process, decomposes it to its constituting

activities, and creates an activity list. This list is later used by the deployer to generate a number of pub/sub messages. Next, the deployer assigns each activity and the associated pub/sub messages to a set of worker nodes.

The worker nodes can execute all the supported BPEL activities, using the received pub/sub messages and the deployed activities descriptions. In our orchestration engine each business process element, such as a BPEL activity, has a corresponding worker node which is an overlay node with pub/sub capabilities. Generally, a worker waits for its predecessor activities to complete by subscribing to such an event, then it executes its deployed activity, and finally it triggers the successor activities by publishing a completion event. As a result, the process execution is event-driven and naturally distributed.

The deployer and worker nodes handle the interactions with the external clients by using services that translate between Web service protocols (such as SOAP over HTTP) and pub/sub formats. This way, the activities in a BPEL process can invoke and be invoked by external Web services.

Operation Overview

Our orchestration engine has five phases of operation: a) startup phase, b) deployment phase, c) execution phase, d) redeployment phase, and e) undeployment phase. In the following, we describe each phase shortly.

During the startup phase, the deployer node acts as an entry point for new nodes to join into the DHT network. If we have multiple deployers, we select one of them (called the bootstrap deployer, as shown in Figure 4.1) to play this role, using a leader selection algorithm [HX07, HH06, GHS83]. The joining nodes are registered as worker or deployer nodes and send their ids to the bootstrap deployer. Moreover, the worker nodes periodically send their utilization status to the deployer. If we have multiple deployer nodes, the above information is replicated among them. In the rest of this section we assume that we have a single deployer.

In the deployment phase, a BPEL process is deployed into a number of worker nodes. This is achieved by the deployer node. The latter using the information gathered after the nodes registration, selects a number of worker nodes with low utilization. Then using the process activity list it assigns the activities to nodes with low utilization. After this step finishes, it sends to the selected worker nodes their assigned activity and a list with associated pub/sub messages. The selected workers complete the deployment of the BPEL activities by publishing these pub/sub messages. With these messages the worker nodes build up the inter-worker dependencies and render the BPEL process ready to execute.

In the execution phase, the deployed business process can be invoked by an external client that communicates with the deployer node. The latter translates the invocation request into a publication message that specifies the process identifier and other required information. When this publication is sent, it triggers the worker node that is responsible for the first activity (e.g. a <receive> activity). The latter executes its activity, publishes a new process instance id, and triggers the successor activity with the proper

publication messages. The same procedure is followed by all the worker nodes that participate in the same process instance; they execute and trigger one another using pub/sub messages in an event-driven manner, until the process instance completes, or a failure happens.

In the redeployment phase, the deployer node re-assigns the BPEL process activities to the workers, based on the workers utilization status. For example, if a worker becomes overloaded or leaves the overlay, the deployer is activated and re-assigns its activities to a different worker with low utilization.

Finally, in the undeployment phase, the deployer receives a request from an external client and undeploys the process from all the related worker nodes. Additionally, it terminates any running process instances.

4.1.1 Deployer Architecture

The architecture of the deployer node, as shown in Figure 4.2, consists of six main components:

1. the *Web Service Gateway* that interacts with external clients
2. the *parser* that transforms the BPEL process to the process activity list
3. the *activity deployment engine* that assigns activities to worker nodes
4. the *subscription matching engine* that matches incoming publications with stored subscriptions
5. the *Publish/Subscribe Gateway* that sends and receives publications and subscriptions from the DHT overlay.
6. the *Deployer cache* that stores subscription messages that are used for pub/sub matching.
7. the *Deployer database*, shown in Figure 4.3, that stores information about the clients, the deployed processes, the running process instances, and the worker nodes.

The *Web Service Gateway* service converts pub/sub messages to SOAP/HTTP messages and vice versa. It provides the functionality for invoking outside Web services and it gives the ability to an outside client to invoke the business process. The Web Service Gateway service is implemented on top of a HTTP server and servlet container (such as Apache Tomcat[2]). In short, the Web Service Gateway is the component which receives deployment and undeployment requests from the clients and sends the process output response back to the clients. Furthermore, each time it receives an incoming request from an external client, it stores the client's address into the *Clients* Table.

[2] http://tomcat.apache.org/

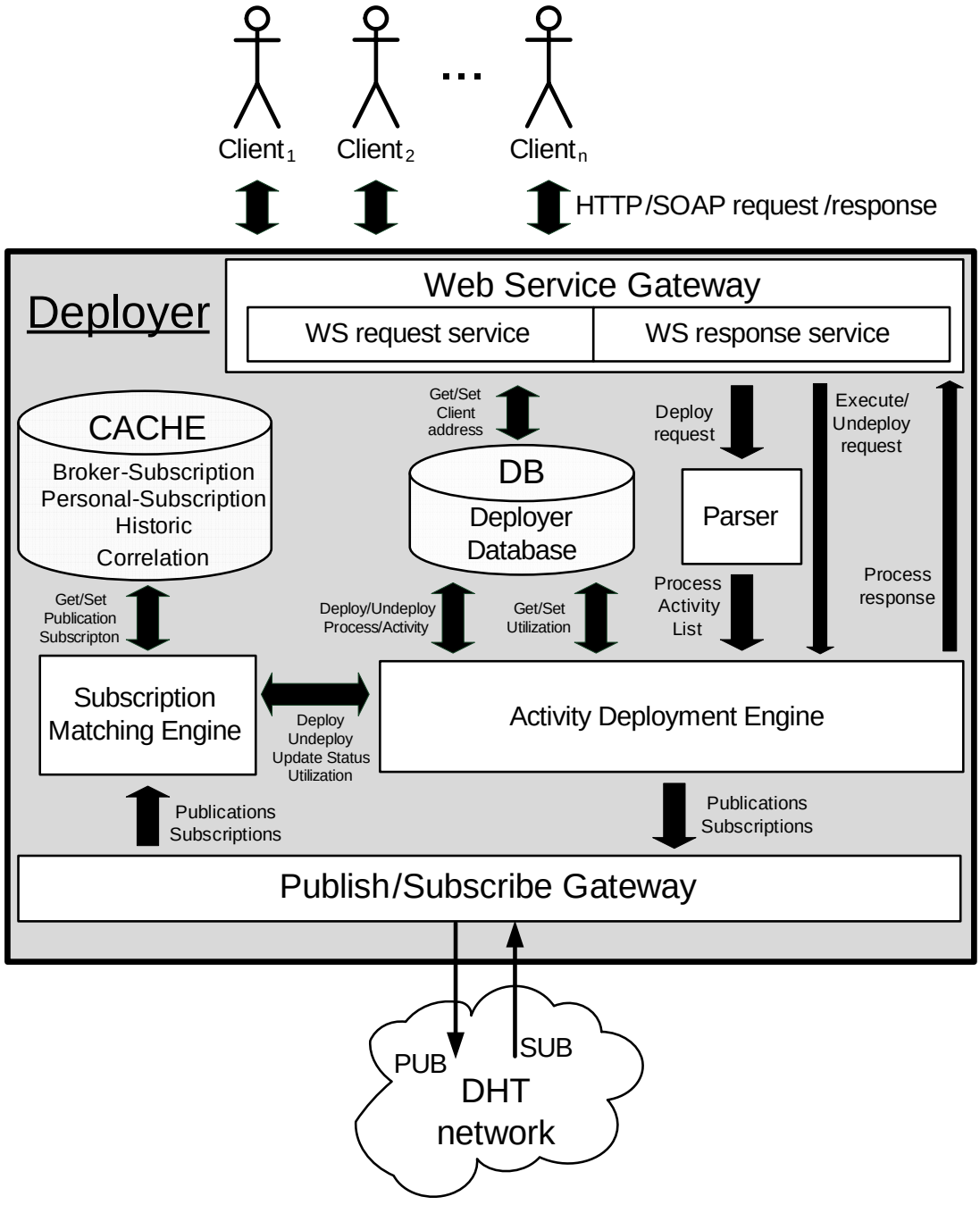


Figure 4.2: Architecture of the deployer node. The deployer node that receives requests from external clients and deploys, undeploys, and triggers the execution of BPEL processes. It can receive deployment, or process instance requests from external clients and it can also send reply messages to the original requesters.

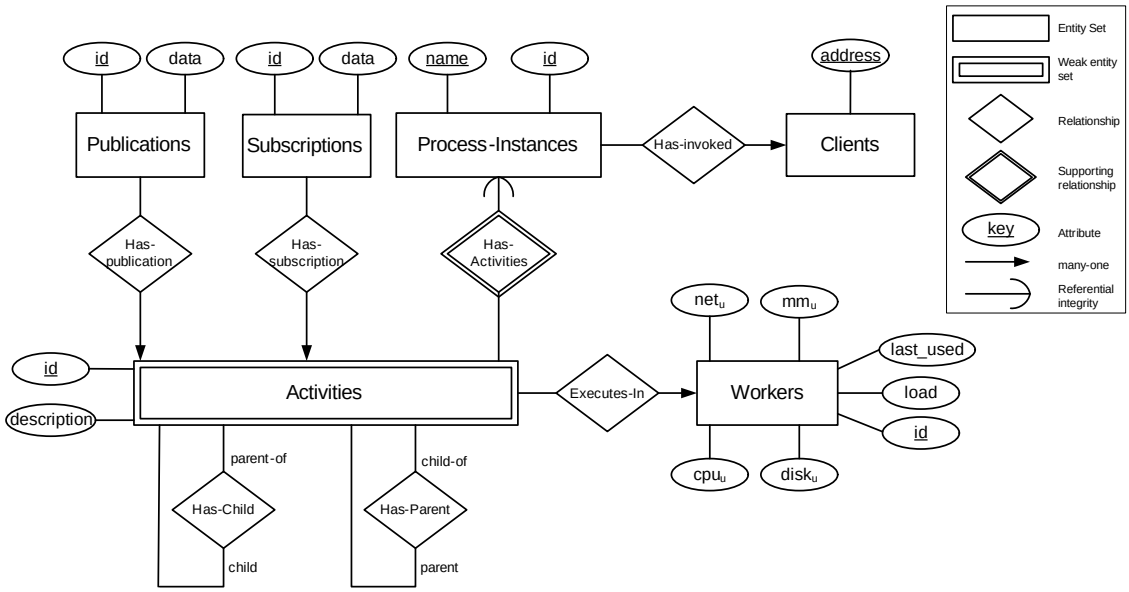
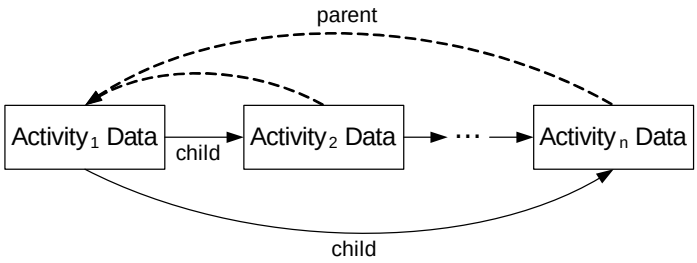


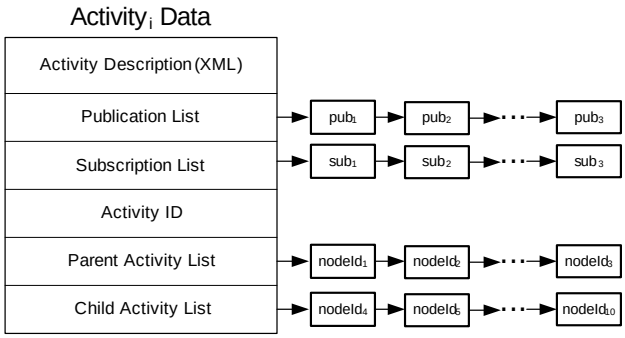
Figure 4.3: E/R model of the deployer node's database.

The *parser* receives as input a compressed bundle containing the BPEL process and the associated WSDL files. This component uncompresses the bundle and parses the contained files, producing a process activity list, as shown in Figure 4.4(a). This list describes all the process activities and defines the data and control flow dependencies among them. Each node in the list represents a BPEL activity and has the structure shown in Figure 4.4(b). As presented it contains the activity's XML description, its unique id, the ids of the parent and child activities, a publication list with the messages that this activity must publish after it is executed, and a list with subscriptions that must be sent during the deployment phase.

The *activity deployment engine* creates a process map that associates process activities with specific worker nodes. This is succeeded by using the information contained into the *Workers* Table and the process activity list. The association is based on the utilization information that the worker nodes send to the deployer. There are two methods for assigning process activities to worker nodes: a) *per-process deployment* and b) *per-instance deployment*. In per-process deployment, for every process instance the deployer uses the same worker nodes for the same activities. Thus, the deployer creates a per-process map as shown in Figure 4.5(a), where each activity belongs to a specific node id. On the other hand, in per-instance deployment, for every process instance the deployer assigns the process activities to different nodes, based on the current utilization status of the worker nodes. Thus, the deployer creates the per-instance map shown in Figure 4.5(b). As illustrated, each process is associated with a number of process instances.



(a) process activity list



(b) activity data

Figure 4.4: Data structures produced after the BPEL process parsing.

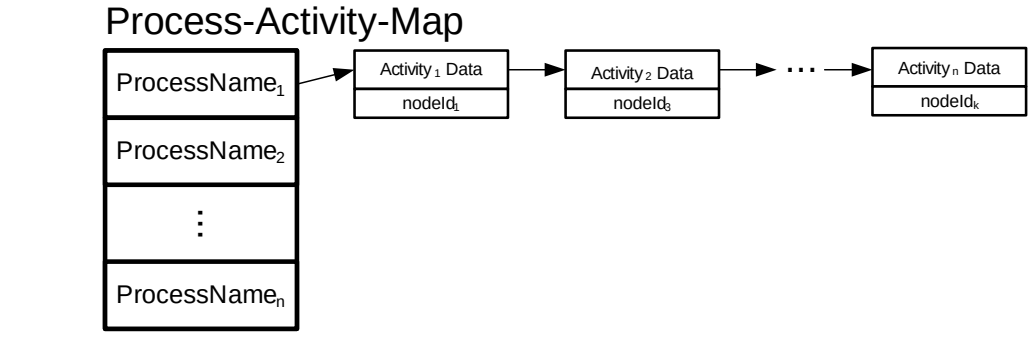
The *subscription matching engine* matches publications with stored subscriptions. The latter are stored in a tree-like structure in the *personal subscription cache* (that stores subscription that belong to the node) and the *broker subscription cache* (that stores subscription that belong to other nodes). In the last case, the deployer acts as a broker node for the pub/sub overlay. The *subscription matching engine* can also match subscriptions that are composite or historic. The composite subscriptions are composed from multiple simple subscriptions and are matched using the correlation trees shown in Figure 4.6. When all the subscriptions that are represented by nodes in the tree are matched, the whole tree evaluates to true. This means that the composite subscription is matched. For the historic subscription (i.e. subscription that are published after their matching publications have been send), we use the internal trees of the historic-cache.

Finally, the *Publish/Subscribe Gateway* sends subscription and publication messages from the activity deployment service to the pub/sub overlay. It also receives pub/-sub messages from the network and delivers them to the subscription matching engine.

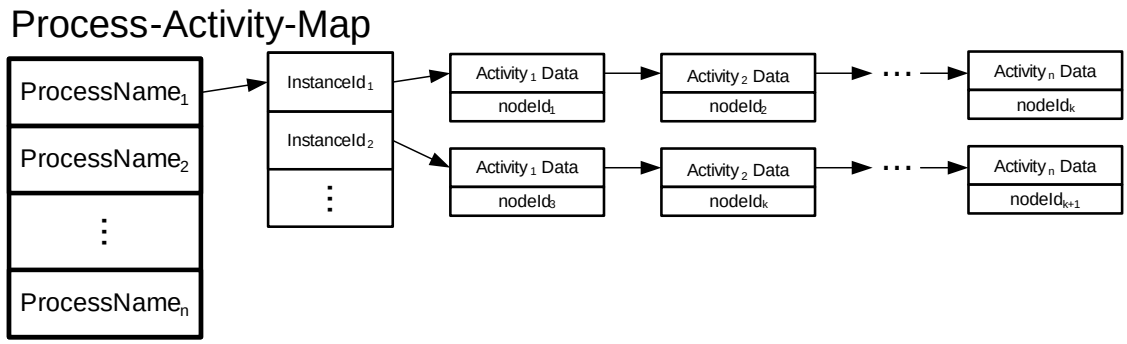
4.1.2 Worker Architecture

The architecture of the worker node, as shown in Figure 4.7, consists of four main components:

1. the *Web Service Gateway* that interacts with external clients



(a) Per-process deployment



(b) Per-instance deployment

Figure 4.5: Deployer node data structures.

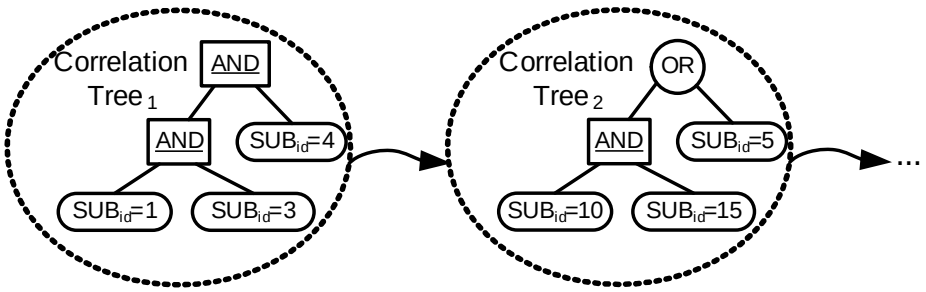


Figure 4.6: Composite subscription list. Each node contains a matching tree.

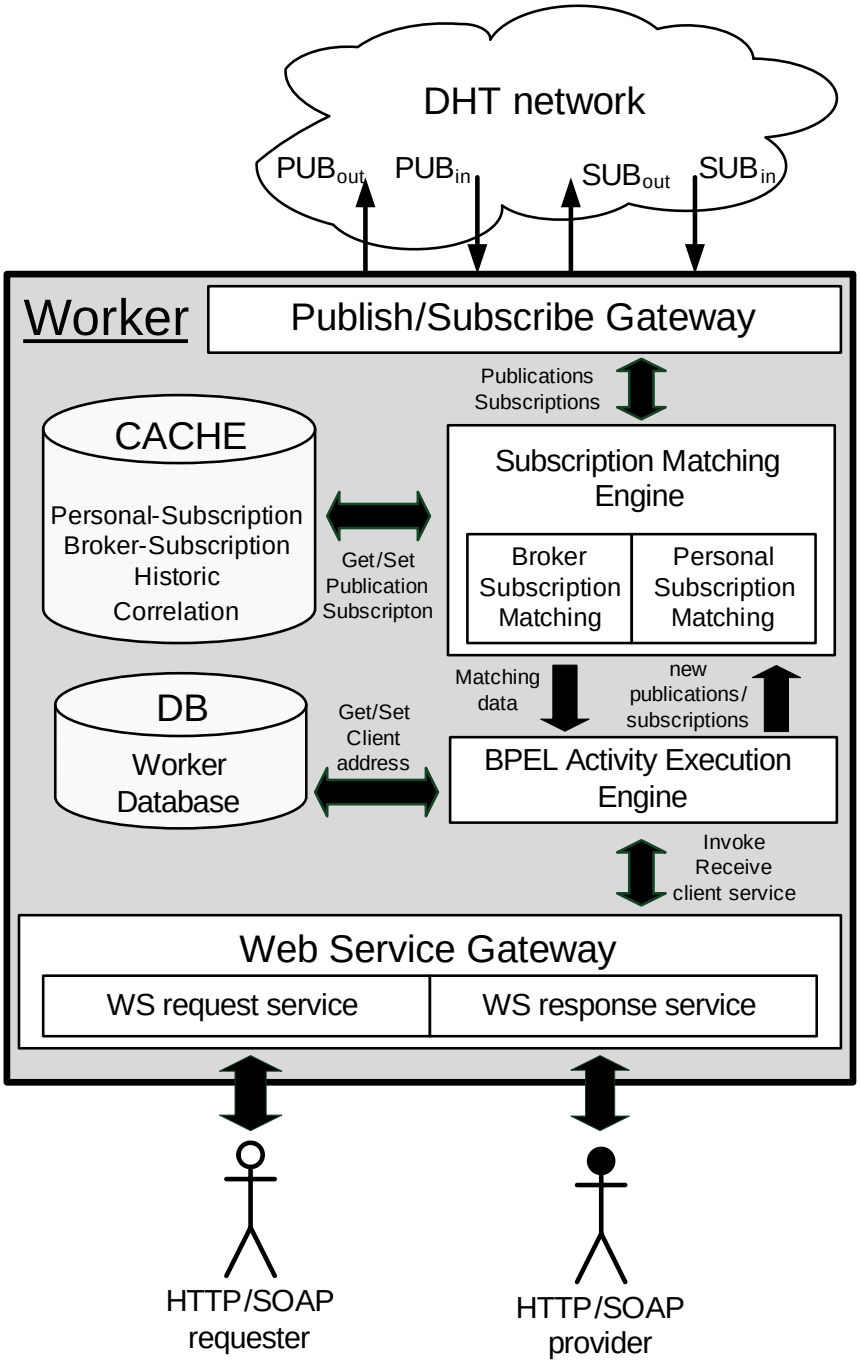


Figure 4.7: The worker node executes BPEL activities and publishes pub/sub messages to transfer data and trigger the execution of the next activity. Furthermore, it can communicate with external clients when executing a <receive>, <invoke>, or <reply> activity.

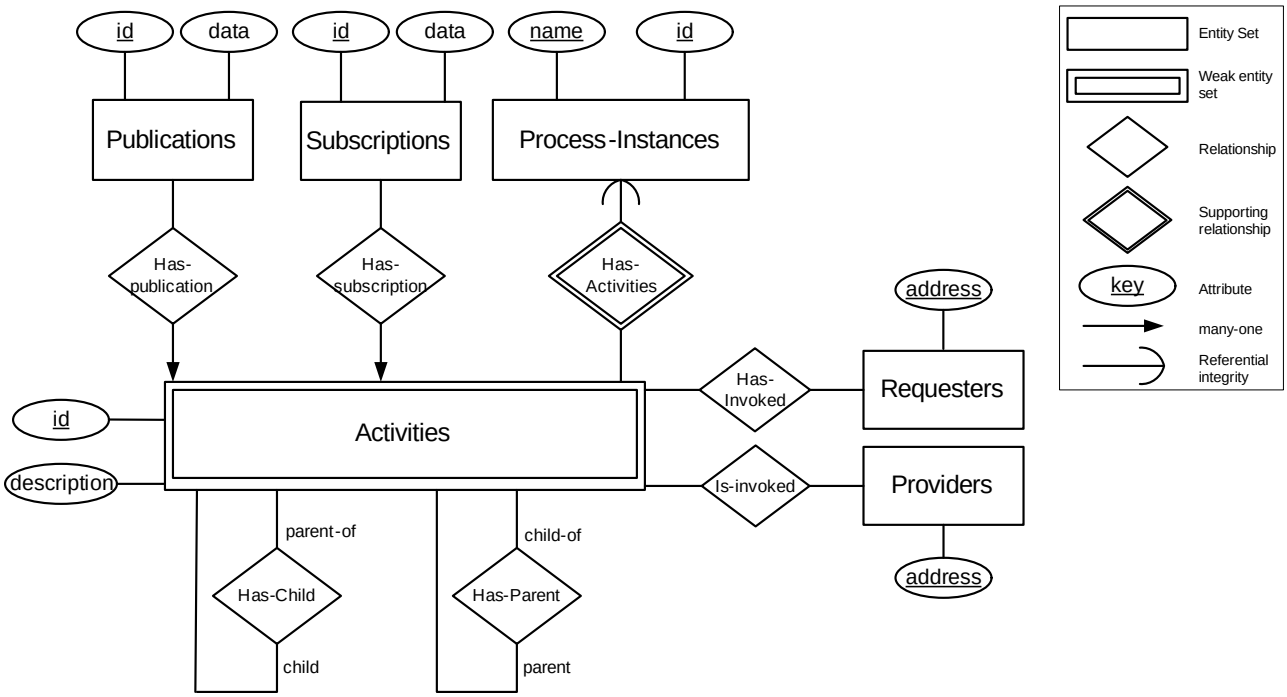


Figure 4.8: E/R model of the worker node's database.

2. the BPEL *activity execution engine* that processes BPEL activities

3. the *subscription matching engine* that matches incoming publications with stored subscriptions

4. the *Publish/Subscribe Gateway* that sends and receives publications and subscriptions from the DHT overlay.

5. the *Worker cache* that stores subscription messages that are used for pub/sub matching.

6. the *Worker database*, shown in Figure 4.8, that stores information about the deployed activities and the external requesters/providers.

The *Publish/Subscribe Gateway* and the subscription matching engine provide the same functionality as those in the deployer node. The BPEL *activity execution engine* processes the execution of the supported BPEL activities, as illustrated in Table 4.1. When the activity executor is triggered and receives the required variable data from the incoming publication messages, it evaluates any conditions or timers and executes the appropriate operations/transformations. Finally, it publishes the updated variable data or any information related to the following activity, using publication messages sent via the *Publish/Subscribe Gateway*.

Basic Activities	
Activity	*Description*
receive	Blocking wait for a message to arrive
reply	Respond to a synchronous operation
invoke	Synchronous or asynchronous Web service call
assign	Manipulate state variables
exit	Terminate a process instance
empty	An empty activity
wait	Delay execution for a duration or deadline

Structured Activities	
Activity	*Description*
sequence	Sequential execution of a set of activities
while	Looping constructs
if	Conditional execution based on instance state
pick	Conditional execution based on events
flow	Concurrent execution

Table 4.1: ADORE basic and structured activities.

The *Web Service Gateway* is the same as the service in the deployer node. It converts publish/subscribe messages to SOAP/HTTP messages and vice versa. It provides the functionality of invoking external Web services, and gives the ability to an external client to invoke the business process. Therefore, it is used by the worker node to implement the <receive>, <invoke>, and <reply> activities.

4.2 Publish/Subscribe over DHT

In a content-based pub/sub system, a subscription is a conjunction of predicates over one or more attributes, while each predicate specifies a range of values for the attribute. A publication is a set of equalities over a set of attributes. In such a system, a subscriber marks its interests using subscriptions and it is notified with matching publications. To accomplish this functionality on a DHT network, we need to resolve the following key problems:

1. How to model the publication and subscription messages to cover our needs?
2. Given a subscription, in which overlay node(s) should we store it?
3. Given a publication, which overlay node(s) must we query to find the matching subscriptions?

4. How can we reduce the pub/sub traffic to provide efficient event dissemination?

5. How can we reduce the subscription storage requirements by applying filter covering and merging techniques?

In the next sections, we describe how our system addresses the above questions.

4.2.1 Publish/Subscribe Model

Publication Model

Definition 1 *An attribute A_i is a tuple $[name, type, value]$. The $name$ is a string value. The $type$ can be an integer, real, or string. The $value$ defines the domain value taken by the given attribute. The domain values belong to a range $[min, max]$ based on the attribute's $type$.*

Definition 2 *A publication pub is defined as a conjuction of attributes: $pub = [A_1 \wedge A_2 \wedge \ldots \wedge A_n]$, where $A_i = [name_i, type_i, value_i]$ for $i \in [1, n]$ is an attribute.*

The class diagram of our implementation is shown in Figure 4.9. As illustrated, a publication message consists of a set of attributes (*attributeSet*), where each attribute is defined as a tuple (*name, datatype, value*). Furthermore, the publication has the unique node identifier (*srcNodeId*) of the publisher node, a timestamp that marks the creation of the publication message (*timeStamp*), a data member to carry additional data (*bodyData*), and a boolean data member (*storePub*) that is set to true when we want to store the publication message into the historic cache of the broker node. This way the (*storePub*) field is used by our engine to match a current publication with a subscription that will arrive later in the network.

Moreover, the figure also shows the deploy activity publication (*DeployActivityPublication*), that is used during the activities deployment. It contains a list of publications (*publicationsList*) that are associated with the activity. During the deployment phase, the deployer node sends *DeployActivityPublication* messages to the worker nodes, that are responsible for executing the process activities.

Subscription Model

Definition 3 *A predicate P_i is a tuple $[name, type, operation, value]$. The $name$ is a string value. The $type$ can be an integer, real, or string. The operation is one of the supported operations (such as: $=$, $\neq$, $<$, $>$, $\leq$, $\geq$, any, starts, ends, contains, inner, not inner). The $value$ is the domain value taken by the given predicate. The domain values belong to a range $[min, max]$ based on the predicate's $type$.*

Definition 4 *A subscription sub is defined as a conjunction of predicates: $sub = [P_1 \wedge P_2 \wedge \ldots \wedge P_n]$, where $P_i = [name_i, type_i, operation_i, value_i]$ for $i \in [1, n]$ is a predicate.*

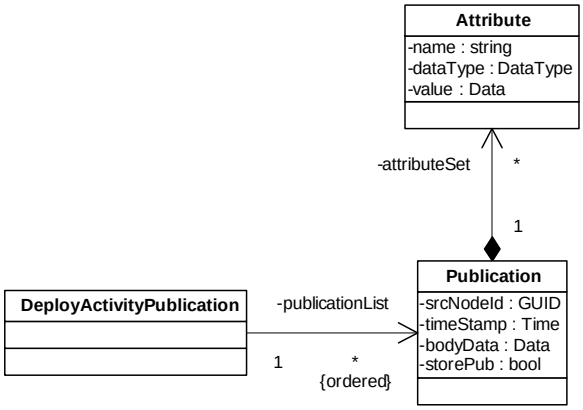


Figure 4.9: Class diagram showing the data members of an attribute, a publication, and a deploy activity publication message.

Definition 5 *A composite subscription* $compSub$ *is defined as:* $compSub = [sub_1 \wedge sub_2 \wedge \ldots \vee sub_n]$*, where each* sub_i *for* $i \in [1, n]$ *is a subscription.*

The class diagram of our implementation is shown in Figure 4.10. A subscription contains the unique node identifier of the subscriber node (*srcNodeId*), a timestamp (*timeStamp*) that marks the creation of the subscription message, and a unique identifier (*subId*) that marks the specific subscription. Furthermore, it has a list of predicates (*predicateSet*), where each predicate is a tuple (*name, datatype, operation, value*). Each predicate can be matched with a specific attribute using the *matchPredicate()* method.

A composite subscription (*CompositeSubscription*) contains a list of subscriptions (*subscriptionList*), is tagged with its creation time (*timeStamp*), and also has the subscriber's id (*srcNodeId*). Additionally, the composite subscription has a correlation tree (*CorrelationTree*) that defines how the individual subscriptions are related i.e. using AND or OR operators. Thus, using a composite subscription we can define complex subscriptions like: $compSub$ = sub_a OR (sub_b AND sub_c), with sub_a = {subId=5, ["name",string, equals, "Jerry Lee"]}, sub_b = {subId=10, ["surname", string, equals, "Lewis"]}, and sub_c = {subId=15, ["date", int, equals, 1935]). This composite subscription produces the Correlation Tree$_2$ shown in Figure 4.6.

Furthermore, Figure 4.10 shows the deploy activity subscription (*DeployActivitySubscription*), that contains of a list of composite subscriptions (*compositeList*). During the deployment phase, the deployer node sends *DeployActivitySubscription* messages to the worker nodes, that are responsible for executing the process activities.

Also, there are a couple of special data types that are used by the previous classes. As shown in Figure 4.11(a), *Data* is a generic type that is used to define the real data that carries the attribute, the predicate, and the publication message.

The *GUID* shown in Figure 4.11(b) is a 16-byte (128-bit) number that is represented as a 32-character hexadecimal string, such as 21EC-2020-3AEA-1069-A2DD-080-02B3-309D. The *GUID*, is used for producing unique node and message identifiers. As the

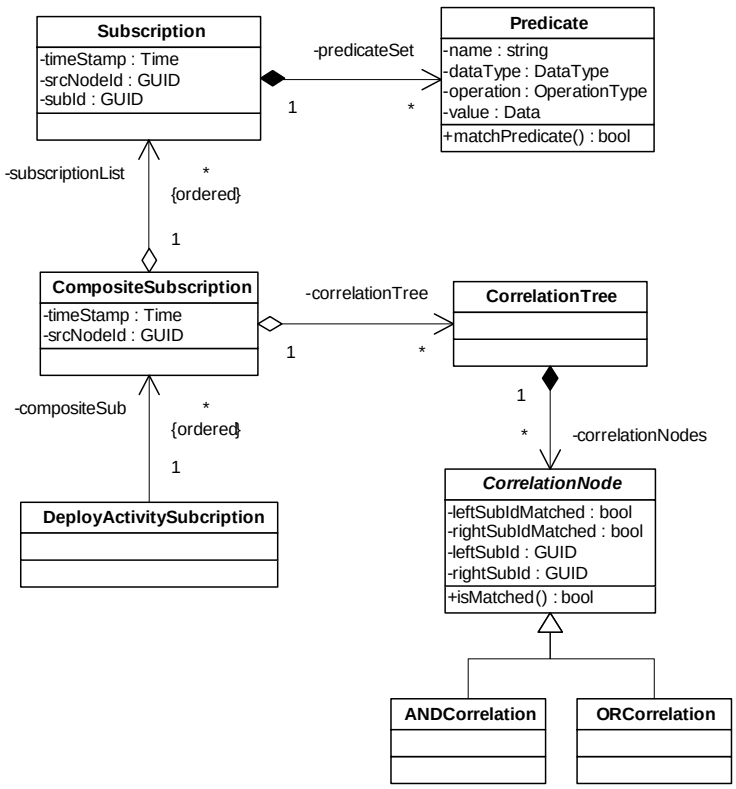
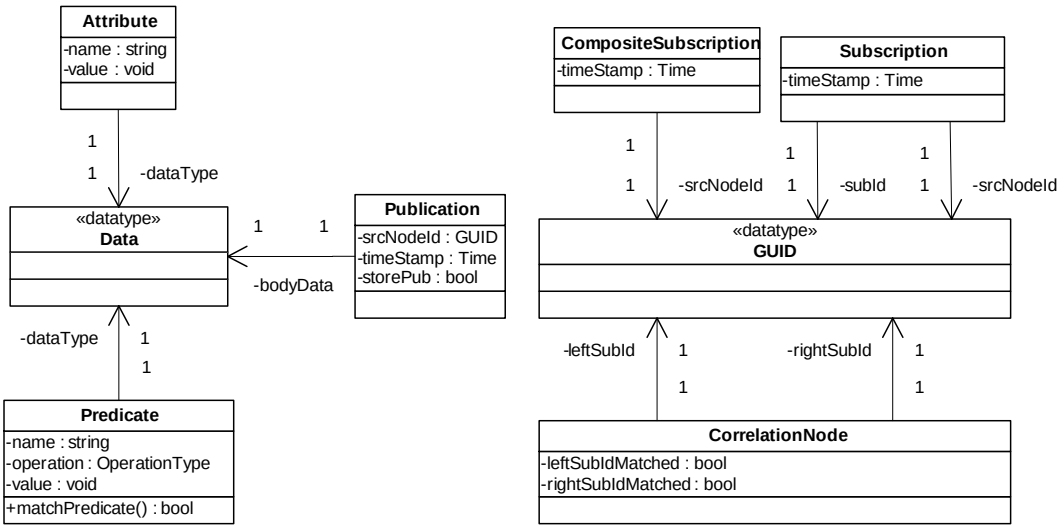


Figure 4.10: Class diagram showing the data members of a predicate, a subscription, a composite subscription, and a deploy activity subscription message.



(a) data class diagram (b) GUID class diagram

Figure 4.11: Class diagram showing the data members of the *Data* and *GUID* classes.

total number of unique keys is 2^{128} the probability of the same number being generated randomly twice is negligible.

The *DataType* (Figure 4.12(b)) defines the type of data that the attribute or the predicate refers to; this can be integer, real, or string. Furthermore, the *DataType* defines the supported operations. If the *DataType* is string the supported operations are: less, greater, equal, not equal, any, starts, ends, contains, inner, and not inner. While if the *DataType* is real or integer we have the operations: less, greater, equal, not equal, any, less equal, greater equal.

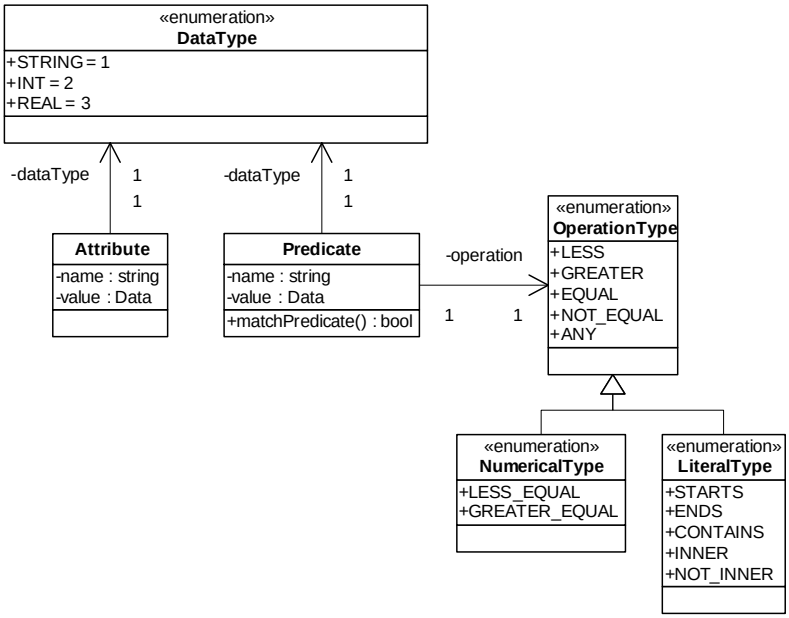
Finally, *Time* as illustrated in Figure 4.12(b), is used to provide timing information and is implemented as a wrapper around the java Date class.

Filter Model

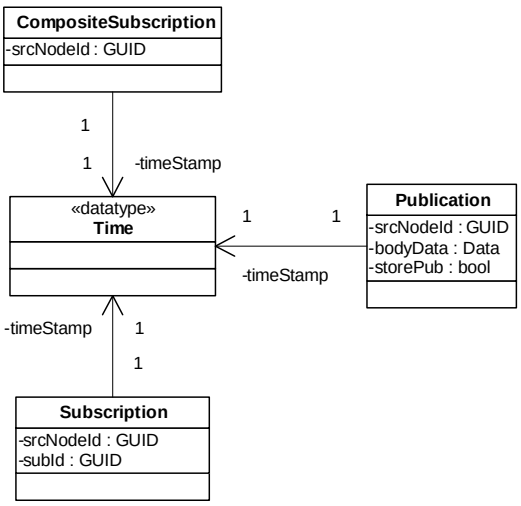
We define as a filter a system subscription that is produced using the predicates of the original subscriptions. A filter consisting of a single atomic predicate is an attribute filter or constraint. Filters that are derived from attribute filters by combining them with boolean operators are compound filters. A compound filter that is a conjunction of attribute filters is called conjunctive filter. In our model we use only conjunctive filters. In the following, with the term filter we will refer to a conjunctive filter.

Definition 6 *An attribute filter f is defined as a simple contraint with only one predicate: $f = [P]$, where $P = [name, type, operation, value]$ is a predicate.*

Definition 7 *An filter F is defined as a conjunction of attribute filters: $F = [f_1 \wedge f_2 \wedge \ldots \wedge f_n]$, where $f_i = [name_i, type_i, operation_i, value_i]$ for $i \in [1, n]$ is a predicate.*



(a) operation type class diagram



(b) time class diagram

Figure 4.12: Class diagram showing the data members of the *DataType* and *Time* classes.

A filter F can be also seen as a stateless boolean function that is applied to a publication, i.e. $F(pub) \rightarrow \{true, false\}$. A publication matches F if $F(n)$ evaluates to true. Consequently, the set of matching publications $P(F)$ is defined as $\{pub|F(pub) = true\}$. Two filters F_1 and F_2 are identical written $F_1 \equiv F_2$, if and only if $P(F_1) = P(F_2)$. Moreover, they are overlapping, denoted by $F_1 \sqcap F_2$, iff $P(F_1) \cap P(F_2) \neq \emptyset$.

Subscription-Publication Matching Model

A matching algorithm tests a given publication against all filters and subscriptions to determine the set of the matched ones. This implies that the same predicate may be evaluated many times. Algorithm 1 presents a commonly used naive matching algorithm based on the idea of the predicate counting [YGM94, PFLS00]. We propose a solution based on a more elaborate approach that is described in section 4.2.4.

Algorithm 1: *Naive Matching Algorithm*

Require: pub = {srcNodeId, $[A_1, A_2, \ldots, A_n]$} //publication message
1: **for all** $A_i \in pub$ **do**
2: **for all** filter f in the routing table **do**
3: **if** $f(pub) = true$ and f has all the attributes of pub **then**
4: add f to matchedSet
5: **end if**
6: **end for**
7: **end for**

Covering Model

The goal of covering-based routing is to remove redundant subscriptions from the network, to force the nodes maintain a compact routing table, and to reduce the network traffic. The concept of covering includes *predicate covering* and *filter covering*. In short, using this method we do not need to store subscriptions that are already covered by existing subscriptions (thus will be matched by the same publications).

Definition 8 *For predicate* $P_i = [name_i, type_i, operation_i, value_i]$ *and predicate* $P_j = [name_j, type_j, operation_j, value_j]$, *we define that* P_i *covers* P_j, *denoted as* $P_i \succeq P_j$, *if and only if* $name_i = name_j$ *and all attribute-value pairs matching* P_i *also match* P_j.

Definition 9 *A filter* F_i *covers another filter* F_j, *denoted* $F_i \succeq F_j$, *if the publication set matching* F_j *also matches* F_i, *that is* $P(F_i) \supseteq P(F_j)$.

Definition 10 *Assuming two filters* $F_i = [f_1^i \wedge f_2^i \wedge \cdots \wedge f_n^i]$ *and* $F_j = [f_1^j \wedge f_2^j \wedge \cdots \wedge f_m^j]$ *that are conjunction of attribute filters with* $n \leq m$, *we define that filter* F_i *covers filter* F_j *denoted as* $F_i \sqsupseteq F_j$, *if and only if* $\forall f_k^i$ *with* $k \in [1, n], \exists f_l^j$ *with* $l \in [1, m] : f_k^i \succeq f_l^j$.

Merging Model

The merging technique is used for further minimizing the routing table size and the network traffic overhead in a content-based network. It is an extension of covering that replaces a number of subscriptions with a more general subscription. If sub_a and sub_b have no covering relations but largely overlap with each other, they can be merged into a more general subscription sub_m.

A new filter F_M, which covers the original filters $(F_1, F_2, \ldots, F_n)$, that is $P(F_M) \supseteq P(F_1) \cup P(F_2) \cup \ldots \cup P(F_n)$, is called a merger of $F_i (i = 1, \ldots, n)$. There are two types of mergers:

1. F_M is a *perfect merger*, if $P(F_M) = P(F_1) \cup P(F_2) \cup \ldots \cup P(F_n)$, that is the publication set of the merger is exactly equal to the union of the publication sets of the original filters.

2. F_M is an *imperfect merger*, if $P(F_M) \supset P(F_1) \cup P(F_2) \cup \ldots \cup P(F_n)$, that is the merger is larger than the filters' union.

A set of conjunctive filters with at most one attribute filter for each attribute can be perfectly merged into a single conjunctive filter with one condition: if for all except a single attribute their corresponding attribute filters are identical and if the attribute filters of the distinguishing attribute can be merged into a single attribute filter [MÖ2]. For example, two filters $F_1 = \{x = 5 \wedge y \in \{2, 3\}\}$ and $F_2 = \{x = 5 \wedge y \in \{4, 5\}\}$ can be merged to $F = \{x = 5 \wedge y \in \{2, 3, 4, 5\}\}$.

4.2.2 Subscription Algorithms

In this section, we examine the methods that are used by our architecture to store the subscriptions to the overlay broker nodes. We have implemented three algorithms with different performance on load balancing, overhead, and network traffic. These algorithms were originally proposed in [ZH05] for Chord DHT. For this reason, we have adapted the original techniques to fit our model and we have extended these algorithms by exploiting the *MSPastry* 's embedded tree geometry.

The subscription storage algorithms are: a) the *Random-Predicate Subscription Algorithm (RP-SA)*, b) the *Proximity-Predicate Subscription Algorithm (PP-SA)*, and c) the *Multi-Predicate Subscription Algorithm (MP-SA)*. In all cases we use a similar two step methodology:

step1 The subscriber chooses an attribute or a number of attributes, extract their names, and produces a key by hashing these names.

step2 Using the produced key, the subscriber calls the $route(key, sub)$ method of the DHT layer that sends the subscription sub to the broker node that has identifier with better proximity to the key.

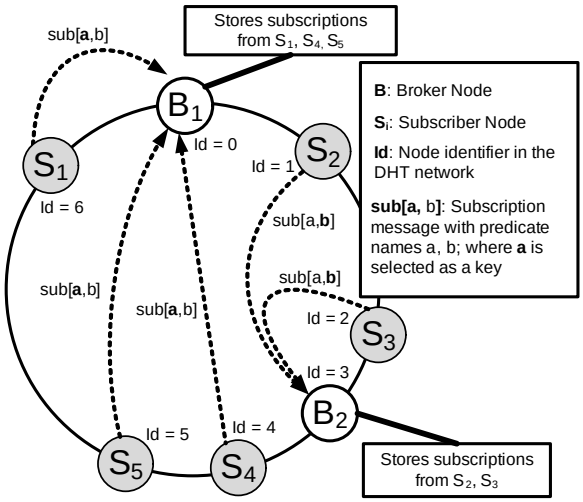


Figure 4.13: Random-Predicate Subscription Algorithm (RP-SA); the key is produced by a randomly selected attribute name. This method has good load balancing characteristics but requires on average a larger number of hops to reach a broker node. The broker B_1 has proximity with the key a, while the broker B_2 has proximity with the key b. Notice that if S_2 used the attribute name a and S_4 used the attribute name b their subscriptions would be stored into a broker with better proximity.

In this section, we also present the corresponding unsubscription algorithms that are used by a subscriber to remove its subscriptions from the broker nodes.

Algorithm 2 presents the *Random-Predicate Subscription Algorithm (RP-SA)*. It selects randomly one attribute from all the available subscription attributes and uses this attribute to produce a key. This method distributes almost randomly the subscriptions to the broker nodes and thus in a large DHT network provides good load balancing. On the other hand, the drawback of this method is that the publications are not always stored to the broker node that is closer to the subscriber. As shown in Figure 4.13 the subscriptions may require more hops than needed to reach the associated broker node.

Algorithm 3 shows the unsubscription process. This method is not efficient as it needs to test all the attribute keys for finding the one that will reach the corresponding broker node. Thus, it requires to send $O(n)$ unsubscription messages, where n is the number of the predicates, that is clearly not optimal.

Algorithm 4 presents a different approach that tries to minimize the average number of hops that are required for a subscription to reach a broker node. In the *Proximity-Predicate Subscription Algorithm (PP-SA)*, all the predicate names are examined and we select the one that produces a key which has larger DHT proximity with a subscriber node. For this purpose, we use the $proximity()$ function that finds the neighbor node which shares the largest common prefix with the given keys (if two or more neighbor nodes have the same longest length of matching prefix with the given key, the method chooses the value that is numerically closest to the subscriber's id).

Algorithm 2: *Random-Predicate Subscription Algorithm (RP-SA).* This method creates a key to route the subscription based on a randomly selected predicate name.

Require: $sub = \{\text{srcNodeId}, [P_1, P_2, \ldots, P_n]\}$ //subscription message
1: $P_i \leftarrow$ choose a random predicate from sub
2: $name_{P_i} \leftarrow$ extract(P_i) //extract the predicate name
3: key = hash($name_{P_i}$) //get the hash value
4: route(key, sub) //route sub to the node that has Id closer to key

Algorithm 3: *Random-Predicate Unsubscription Algorithm (RP-UA)* unsubscribes a subscriber that used the RP-SA algorithm.

Require: $unSub = \{\text{srcNodeId}, [P_1, P_2, \ldots, P_n]\}$ //unsubscription message
1: **for** each attribute $P_i \in unSub$ **do**
2: $name_{P_i} \leftarrow$ extract(P_i) //extract the predicate name
3: $k_i =$ hash($name_{P_i}$) //get the hash value
4: route($k_i, unSub$) //route $unSub$ to the node with Id closer to k_i
5: **end for**

Thus, in this case the keys are selected based on their proximity, while we choose the one that will be stored closer to the subscriber node. This way, the algorithm requires on average lower number of hops for the subscription to reach the appropriate broker node (as shown by Figure 4.14). On the other hand, this method can create unbalanced network load: if few nodes produce many subscriptions, then a small fraction of broker nodes may receive a large number of subscriptions.

Nevertheless, the unsubscription method, presented by Algorithm 5, is optimal as it requires only a single unsubscription message to be sent.

Algorithm 4: *Proximity-Predicate Subscription Algorithm (PP-SA).* This method creates a key to route the subscription using the subscription's predicate name that has better proximity.

Require: $sub = \{\text{srcNodeId}, [P_1, P_2, \ldots, P_n]\}$ //subscription message
1: **for** each predicate $P_i \in sub$ **do**
2: $name_{P_i} \leftarrow$ extract(P_i) //extract the predicate name
3: $k_i =$ hash($name_{P_i}$) //get the hash value
4: **end for**
5: $key =$ proximity($k_1, k_2, \ldots, k_n$) //get the k_i with better proximity to the subscriber's Id
6: route($nearest_{key}, sub$) //route sub to the node with Id closer to $nearest_{key}$

Finally, Algorithm 6 presents the *Multi-Predicate Subscription Algorithm (MP-SA)* that produces a key by hashing all the predicate names of the subscription. The subscription

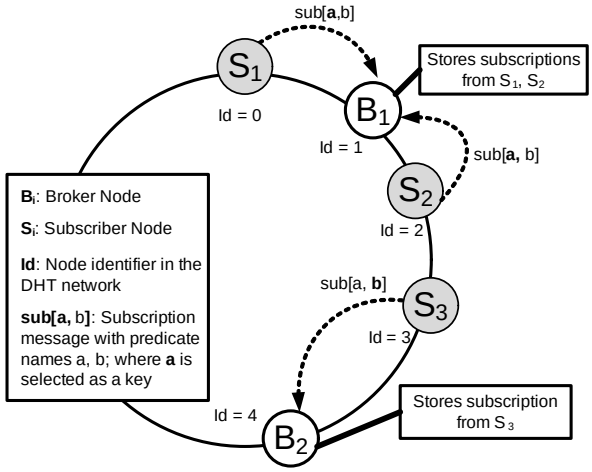


Figure 4.14: Proximity- Predicate Subscription Algorithm (PP-SA); selects the key that has larger proximity with the subscriber's node id. The broker B_1 has proximity with the key a, while the broker B_2 has proximity with the key b. This method has poor load balancing characteristics but requires on average less hops for the subscription to reach a broker node.

Algorithm 5: *Proximity-Predicate Unsubscription Algorithm (PP-UA)* unsubscribes a subscriber that used the PP-SA algorithm.

Require: $unSub = \{\text{srcNodeId}, [P_1, P_2, \ldots, P_n]\}$ //unsubscription message
1: **for** each predicate $P_i \in unSub$ **do**
2: extract the predicate name $name_{P_i}$
3: $k_i = \text{hash}(name_{P_i})$ //get the hash value
4: **end for**
5: $key = \text{proximity}(k_1, k_2, \ldots, k_n)$ //get the k_i with better proximity to the subscriber's nodeId
6: route($key, unSub$) //route $unSub$ to the node with Id closer to key

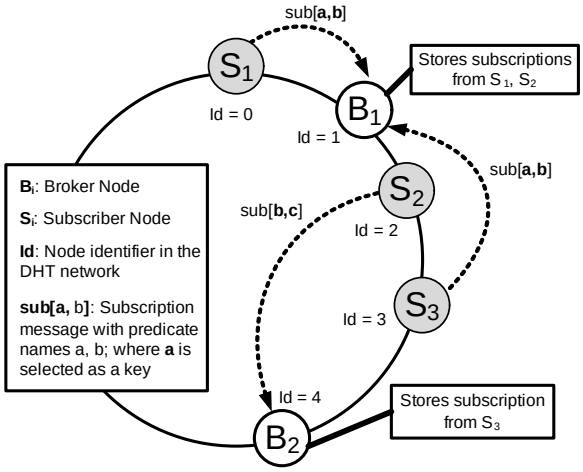


Figure 4.15: Multi-Predicate Subscription Algorithm (MP-SA); selects all the predicate names to produce a key. The subscription is forwarded to the neighbor node that has larger proximity with the key. The broker B_1 has proximity with the key ab, while the broker B_2 has proximity with the key bc. This method's load balancing characteristics depend on the number of subscriptions with the same attribute names.

is stored to the broker node that is numerically closer to the key. The advantage of this method is that synonymous subscriptions (subscriptions that have the same predicate names) are stored to the same broker node. This property can lead to more efficient pub/sub matching when combined with covering and merging techniques, as we will see in section 4.2.5. On the other hand, this method has problems with load balancing: if we have many subscriptions with the same attributes names, then a small number of brokers will store all these subscriptions and will become overloaded (as depicted by Figure 4.15). Nevertheless, the unsubscription method, presented by Algorithm 7, requires only a single unsubscription message to be sent to the proper broker node, which is optimal.

Algorithm 6: *Multi-Predicate Subscription Algorithm (MP-SA).* This method creates a key to route the subscription using all predicate names.

Require: $sub = \{\text{srcNodeId}, [P_1, P_2, \ldots, P_n]\}$ //subscription message
1: $nameSet \leftarrow$ extract($P_1, P_2, \ldots, P_n$) //extract all the predicate names
2: sort all predicate names into $sortedNameSet$
3: $key =$ hash($sortedNameSet$) //get the hash value
4: route(key, sub) //route sub to the node with Id closer to key

Algorithm 7: *Multi-Predicate Unsubscription Algorithm (MP-UA)* unsubscribes a subscriber, that used the MP-SA algorithm.

Require: $unSub = \{\text{srcNodeId}, [P_1, P_2, \ldots, P_n]\}$ //unsubscription message
1: extract all the predicate names into $nameSet$
2: sort all predicate names into $sortedNameSet$
3: $key = hash(sortedNameSet)$ //get the hash value
4: route($key, unSub$) //route $unSub$ to the node with Id numerically closer to key

4.2.3 Publication Algorithms

In this section, we examine the methods that are used by our architecture to query the overlay nodes for finding matching subscriptions. We have implemented two different publication algorithms. These are: a) the *Single-Attribute Publication Algorithm (SA-PA)*, and b) the *Multi-Predicate Publication Algorithm (MA-PA)*. These algorithms are used in par with the previously presented subscription algorithms.

Algorithm 8 shows the *Single-Attribute Publication Algorithm (SA-PA)*, that is used when we store subscriptions using the RP-SA (Algorithm 2), or the PP-SA (Algorithm 4). At first, it examines all the publication's attributes and produces a key for each attribute. Then, the publisher sends multiple copies of the publication, each time using one of the produced keys. This way, the publication can reach all the possible associated broker nodes. This method is clearly not optimal as it produces a number of publication messages equal to the number of the publication attributes. Thus, a single publication message with n attributes must be sent $O(n)$ times.

Algorithm 8: *Single-Attribute Publication Algorithm (SA-PA)*. This method is used to send publications and is combined with the RP-SA and PP-SA algorithms.

Require: $pub = [A_1, A_2, \ldots, A_n]$ //publication message
1: **for all** each attribute $A_i \in pub$ **do**
2: extract the attribute name $name_{A_i}$
3: $k_i = \text{hash}(name_{A_i})$ //get the hash value
4: route(k_i, pub) //route pub to the node with nodeId numerically closer to k_i
5: **end for**

On the other hand, Algorithm 9 presents the *Multi-Attribute Publication Algorithm (MA-PA)*. This method is used when we store subscriptions using the MP-SA (Algorithm 6). This time, the publisher has to send only a single publication message, while the message is propagated to the broker node with better proximity. The key is produced using all the attributes names. Thus, a single publication message with n attributes must be sent $O(1)$ times.

Algorithm 9: *Multi-Attribute Publication Algorithm (MA-PA).* This method is used to send publications and is used in par with the MP-SA algorithm.

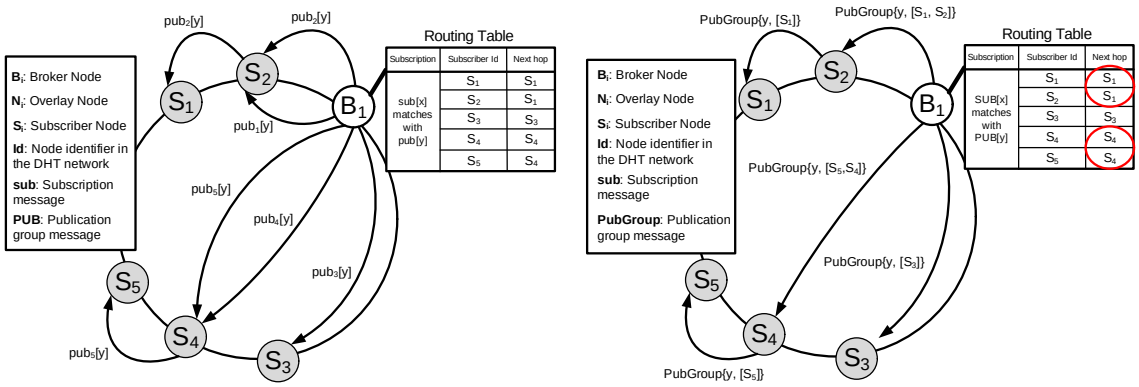
Require: $pub = [A_1, A_2, \ldots, A_n]$ //publication message
1: extract all the attribute names into $nameSet$
2: sort all attribute names into $sortedNameSet$
3: $key =$ hash($sortedNameSet$) //get the hash value
4: route(key, pub) //route pub to the node with Id closer to key

4.2.4 Event Delivery Algorithms

The overhead produced by the event delivery algorithm (as an event we define a matched publication), is crucial for the efficiency of our pub/sub overlay. Figure 4.16(a), presents a naive event delivery algorithm where a publication $pub[y]$ is matched with a subscription $sub[x]$ in broker node B_1. Using the subscription matching engine the broker creates a set with interested subscribers' ids and sends the publication one time to each subscriber. Clearly, this method is not optimal because (as shown in Figure 4.16(a)), many subscribers may be on the same delivery path. Thus, we propose the group deliver algorithm (Algorithm 11), which groups subscribers that belong to the same delivery path into the same group publication message (as shown in Figure 4.16(b)). In this case, the broker B_1 creates a publication group message and sends a single publication for the subscribers S_1 and S_2 that belong to the same delivery path using group delivery. In the example presented in Figure 4.16, the number of the published messages between the two methods is reduced from 7 to 5. In the rest of this section, we describe the publication matching and the group delivery mechanism that is used by our approach.

The first step is the publication matching. Algorithm 10 matches a publication with stored subscriptions in the broker's cache using the algorithm that was originally proposed in [ASS+99]. In this approach, each subscription is a conjunction of elementary predicates and each predicate represents one possible result of an elementary test. An elementary test is a simple operation on one or more attributes of the publication pub. For example, we can have $sub = [P_1]$, P_1=[city, eq, "New York"], and $test_1()$ may be "examine attribute city". Thus, we have $test_1(sub) \rightarrow$"New York". Using the above rules, we construct matching trees as the one shown in Figure 4.17. Each non-leaf node contains a test, and edges from that node represent results of that test. A leaf node l, contains a subscription sub instead of a test. Intuitively, sub is the subscription described by walking the tree from the root to l and taking the conjunction of the elementary predicates.

Then, for every matching subscription we extract it's subscriber id (*srcNodeId*) and find the neighbor node that has closest proximity. This way, the broker creates a map (*routeMap*) that associates neighbor nodes with subscriber ids. This map is delivered to Algorithm 11, which creates and sends to each neighbor in the $routeMap$ a group publication message. The latter contains the publication and a list with ids of the interested subscribers.



(a) naive event delivery (b) group publication delivery

Figure 4.16: The naive event delivery algorithm shown in Figure 4.16(a), matches a publication with a number of subscriptions and sends the publication to each subscriber without examining the path back to the subscriber. Our publication algorithm, shown in Figure 4.16(b), groups the subscribers that belong to the same delivery path and sends one publication for each different delivery path.

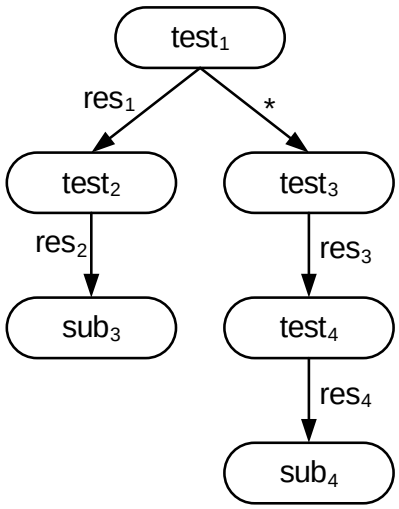


Figure 4.17: Subscription matching tree with a *-edge that represents a don't care test. These edges are necessary when some of the subscriptions are independent of the test.

Finally, each neighbor node that receives a group publication, handles the message using Algorithm 12. This algorithm checks the list of the interested subscriber ids with the nodes own id. If a match is found, the publication is delivered to the application layer and the group publication message is propagated to the remaining subscribers. If no match is found, then the group publication is simply propagated to the neighbor nodes.

Algorithm 10: *Publication Matching* algorithm provides pub/sub matching.

Require: $pub = [A_1, A_2, \ldots, A_n]$ //publication message
1: **for all** matched subscription sub_i **do**
2: $matchedSet.add(sub_i)$ //add subscription in the matched set
3: **end for**
4: **for all** subscription $sub_i \in matchedSet$ **do**
5: extract $srcNodeId_i$ from the sub_i //extract the subscriber's id from the subscription message
6: nb_{id} = proximity($srcNodeId_i$) //get the neighbor node id that is closer to the subscriber's id
7: routeMap.add(nb_{id}, $srcNodeId_i$) //add the tuple $< nb_{id}, srcNodeId_i >$ to the route map
8: groupDeliver(pub, routeMap)
9: **end for**

Algorithm 11: *Group Deliver* algorithm sends a publication group message to a set of interested subscribers.

Require: $pub = [A_1, A_2, \ldots, A_n]$ publication message
Require: routeMap that associates a neighbor's id with a list of subscribers' ids.
1: **for all** entries nb_{id} of routeMap **do**
2: $pubGroup_i \leftarrow pub + routeMap.get(nb_{id})$ //create pubGroup message
3: routeMsg($pubGroup_i$) //send $pubGroup_i$ to nb_{id}
4: **end for**

4.2.5 Filter Covering/Merging Algorithms

A common problem that occurs in all the pub/sub matching algorithms, is that in many cases a small number of broker nodes are responsible for matching the vast majority of the submitted subscriptions. In extreme cases, these nodes become overloaded and fail to handle more publications. In order to solve these problems, we use a filter merging algorithm that substitutes subscriptions with a cover filter as proposed in [MÖ2, Tar07]. We call this two-step process pub/sub table reduction. In the first step, we discover the filter covering and merging candidates in the pub/sub tables using covering and merging tests.

Algorithm 12: *Group Message Propagation* algorithm provides subscription id matching and propagates group publication message to the remaining subscribers.

Require: pubGroup message
1: subIdList ← extractSubscribers($pubGroup$)
2: pub ← extractPublication($pubGroup$)
3: **for all** $srcNodeId_i \in subIdList$ **do**
4: **if** $srcNodeId_i$ matches with current node's id **then**
5: new publication is received
6: **else**
7: find the node nb_{id} that is closer to the $srcNodeId_i$
8: routeMap($nb_{id}, srcNodeId_i$) //update routeMap
9: **end if**
10: **end for**
11: **if** routeMap is not empty **then**
12: groupDeliver(pub, routeMap)
13: **end if**

For the covering we use two tests: a) we employ Algorithm 15 to determine all the filters that cover a given subscription and b) we employ Algorithm 16 to determine all the filters that are covered by a given subscription. For the filter merging we use Algorithm 17 that determines all the possible merging candidates. These are filters that are identical to the given filters in all but a single attribute [MÖ2].

In the second step, the real subscription messages are pushed from the original broker node to its neighbors. For example, in Figure 4.18(a) the broker node B_1 must provide matching for subscriptions $sub[x]$, $sub[y]$, and $sub[z]$. We notice that the publications that match with $sub[y]$ and $sub[z]$ have a common propagation path. Thus, we replace the $sub[y]$ and $sub[z]$ with a filter that covers them and push these subscriptions to the first node in their path, as shown in Figure 4.18(b). This way, B_1 needs only to do filter matching, which consumes less physical resources and does not depend on the number of stored subscriptions. One drawback of this method is that we increase the publication path by one hop for each subscription push.

Algorithm 13 implements the above process by finding subscriptions that have the same propagation path and replaces them with a covering filter. Furthermore, it pushes the real subscriptions to the first node in the propagation path. When a publication is received, we use Algorithm 14. The latter tests for a match with the installed filters or subscription matching trees. If a match with an installed filter is found, then the publication is propagated to the next node in the propagation path. Otherwise, the subscriber ids are inserted into the *matchedSet* and the same group delivery process is followed, as in Algorithm 12. Notice, that in the case of the filter match, the broker node simple re-transmits the received publication message.

Algorithm 13: *Filter Installation* algorithm provides publication routing that reduces the overhead caused by publication matching on the broker nodes.

1: find subscriptions e.g. (x, y, z) that have common propagation path
2: replace subscriptions (x, y, z) with $filter_{x,y,z}$
3: push the subscription x, y, z to the first node on the propagation path

Algorithm 14: *Filtered Publication Matching* algorithm provides filter and publication matching.

Require: $pub = [A_1, A_2, \ldots, A_n]$ publication message
1: **for all** matched subscription sub_i or matched filter $filter_i$ **do**
2: **if** matched $filter_i$ **then**
3: send pub to the next propagation path node
4: **else**
5: $matchedSet.add(sub_i)$ //add subscription in the matched set
6: **end if**
7: **end for**
8: **for all** subscription $sub_i \in matchedSet$ **do**
9: extract $srcNodeId_i$ from the sub_i
10: $nb_{id} = proximity(srcNodeId_i)$ //based on the routing table get the node nb_{id} that is closer to the $srcNodeId_i$
11: routeMap.add(n_i, sid_i) //add sid_i to the route map
12: groupDeliver(pub, routeMap)
13: **end for**

Algorithm 15: *Covering Algorithm*. This method checks whether the filters in the routing table are covering the input subscription. Returns the set of the covering filters.

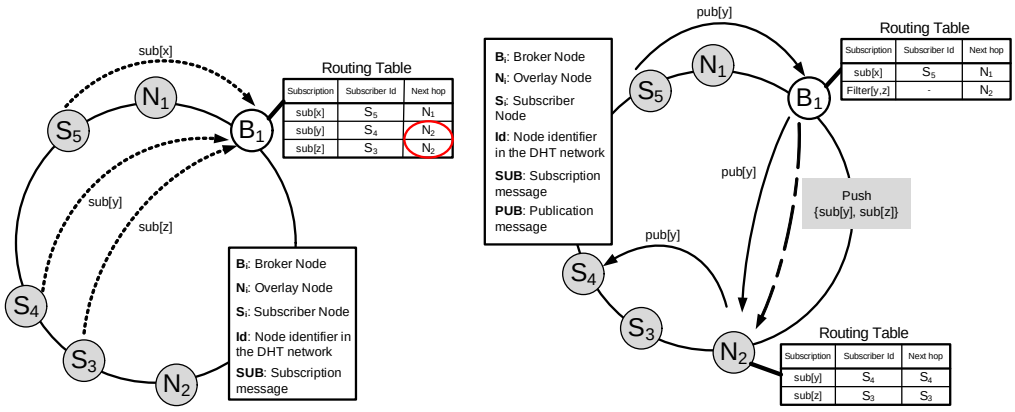
Require: $sub = \{\text{srcNodeId}, [P_1, P_2, \ldots, P_n]\}$ //subscription message
Require: set of all filters F
1: set for each filter in F a counter that is initialized to zero
2: **for all** $P_i \in sub$ **do**
3: **for all** filter $f \in F$ that has a constraint A_i that covers P_i **do**
4: increment the counter for f
5: **end for**
6: **end for**
7: return all filters in F whose counter is equal to their attributes filters.

Algorithm 16: *Covered Algorithm.* This method identifies which filters are covered by the input subscription and returns the set of the covered filters.

Require: $sub = \{\text{srcNodeId}, [P_1, P_2, \ldots, P_n]\}$ //subscription message
Require: set of all filters F
1: set for each filter in F a counter that is initialized to zero
2: **for all** $P_i \in sub$ **do**
3: **for all** filter $f \in F$ that has a constraint A_j that is covered by P_i **do**
4: increment the counter for f
5: **end for**
6: **end for**
7: return all filters in F whose counter is equal to number of attribute filters of sub.

Algorithm 17: *Merging Algorithm.* Returns a set of filters that can be merged together.

Require: filter f_1
Require: set of all filters F
1: set for each filter in F a counter that is initialized to zero
2: **for all** $A_i \in f_1$ **do**
3: **for all** filter $f \in F$ that has a constraint A_j that is identical to A_i **do**
4: increment the counter for f
5: **end for**
6: **end for**
7: return all filters in F whose counter is one smaller that or equal to the number of attribute filters.



(a) Subscription matching without filter merging (b) Subscription matching with filter merging

Figure 4.18: Subscription matching without support for filter merging tends to overload the broker nodes 4.18(a). A solution is to support filter merging and to delegate the computationally intensive task of matching pub/sub messages to neighbor nodes, as shown in Figure 4.18(b).

4.3 Mapping BPEL to Publish/Subscribe Messages

In order to support the decentralized execution of the BPEL activities we must translate the BPEL language constructs into the pub/sub language used by our system. The translation transforms the deployed BPEL process into a behaviorally equivalent set of pub/sub messages that can be processed directly by the deployer and the worker nodes of the DHT overlay. This means that any data and control flow dependencies that exist in a BPEL process must be described via pub/sub interactions.

An overview of how the supported simple and structured BPEL activities in Table 4.1 are translated into pub/sub messages is shown in Figure 4.19 for the simple activities and in Figure 4.20 for the structured ones. The reader can find the details about the structure and the production of those messages in the appendix A. The process of translating the BPEL process description from XML to the according pub/sub messages is carried out by the parser module in the deployer node.

The control flow dependencies are transformed into the exchange of messages among the worker activities. For example, each worker subscribes to a number of publications from the predecessor activity and waits for the proper publication messages. When the worker responsible for the previous activity finishes its execution, it sends publication status messages that trigger the waiting activity. This process continues until the last activity, whereby its related worker finishes its execution.

The data flow dependencies are handled in a similar manner, as the BPEL variables are disseminated into the subscription language. When mapping activities to worker nodes, the involved variables are mapped as part of the deployed subscriptions. After

an activity finishes its execution, the corresponding worker node sends publications that contain the updated variables. Every worker node with an activity that depends on those variables has already made an according subscription. Thus, it will receive the publications with the updated values. Using this approach, there is no need for a centralized entity that will handle all the variables. Therefore, we avoid a possible scalability bottleneck and failure point.

4.4 System Operation

In the following sections we describe in detail the operation of our orchestration engine during its five phases: a) startup, b) deployment, c) execution, d) redeployment, and e) undeployment. Furthermore, we present two unique features of our engine:

- Its ability to use multiple deployer nodes for deploying business processes.
- Its activity deployment mechanism, that exploits the network condition and the agents utilization state to map activities to network nodes.

We must note that our engine is designed to operate with multiple deployer nodes. Nevertheless, to make the presentation easier for the reader, we start by describing the operation phases using a single deployer (as in most cases the required steps remain the same) and then we elaborate for the case of multiple deployers by filling out the missing details.

4.4.1 Startup Phase

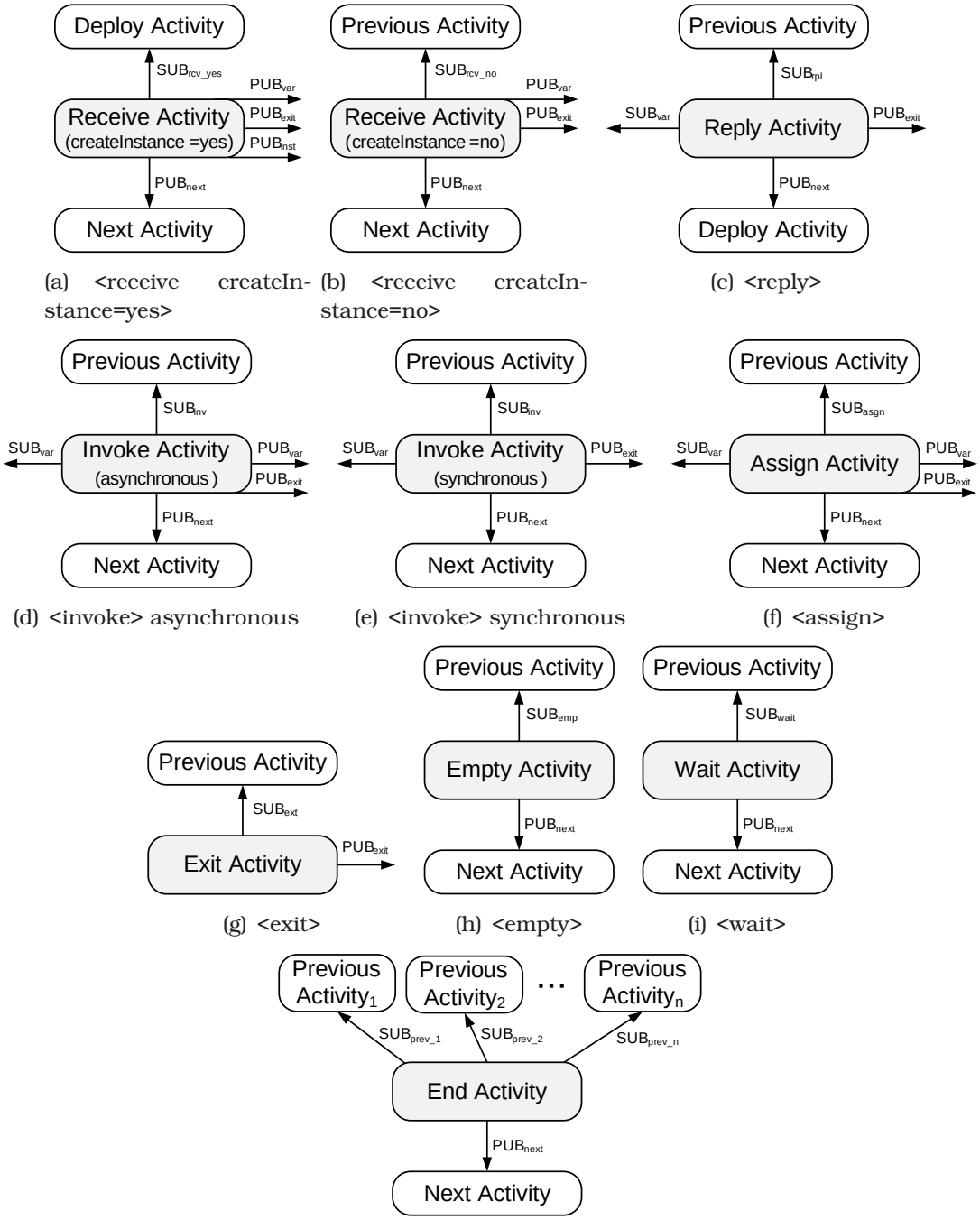
During the startup phase, the worker nodes join the DHT network and register to the deployer node. The deployer node subscribes to the workers' publications, by registering its interest on the utilization messages. Table 4.2 presents the deployer messages, while Table 4.3 presents the worker messages.

On their behalf, the worker nodes subscribe to publications from the deployer node, that carry information about the deployed activities. Figure 4.21, presents all the interactions that happen during the startup phase.

Single Deployer Node

The deployer's role is critical for the engine's correct and efficient operation. For this reason, this node should remain stable and must always participate into the DHT network[4]. The deployer serves three main purposes:

[4]As we cannot tolerate any fault in this node, in a real system implementation with only one deployer we must use a cluster of machines that facilitate the same replicated information.



(a) <receive createInstance=yes>

(b) <receive createInstance=no>

(c) <reply>

(d) <invoke> asynchronous

(e) <invoke> synchronous

(f) <assign>

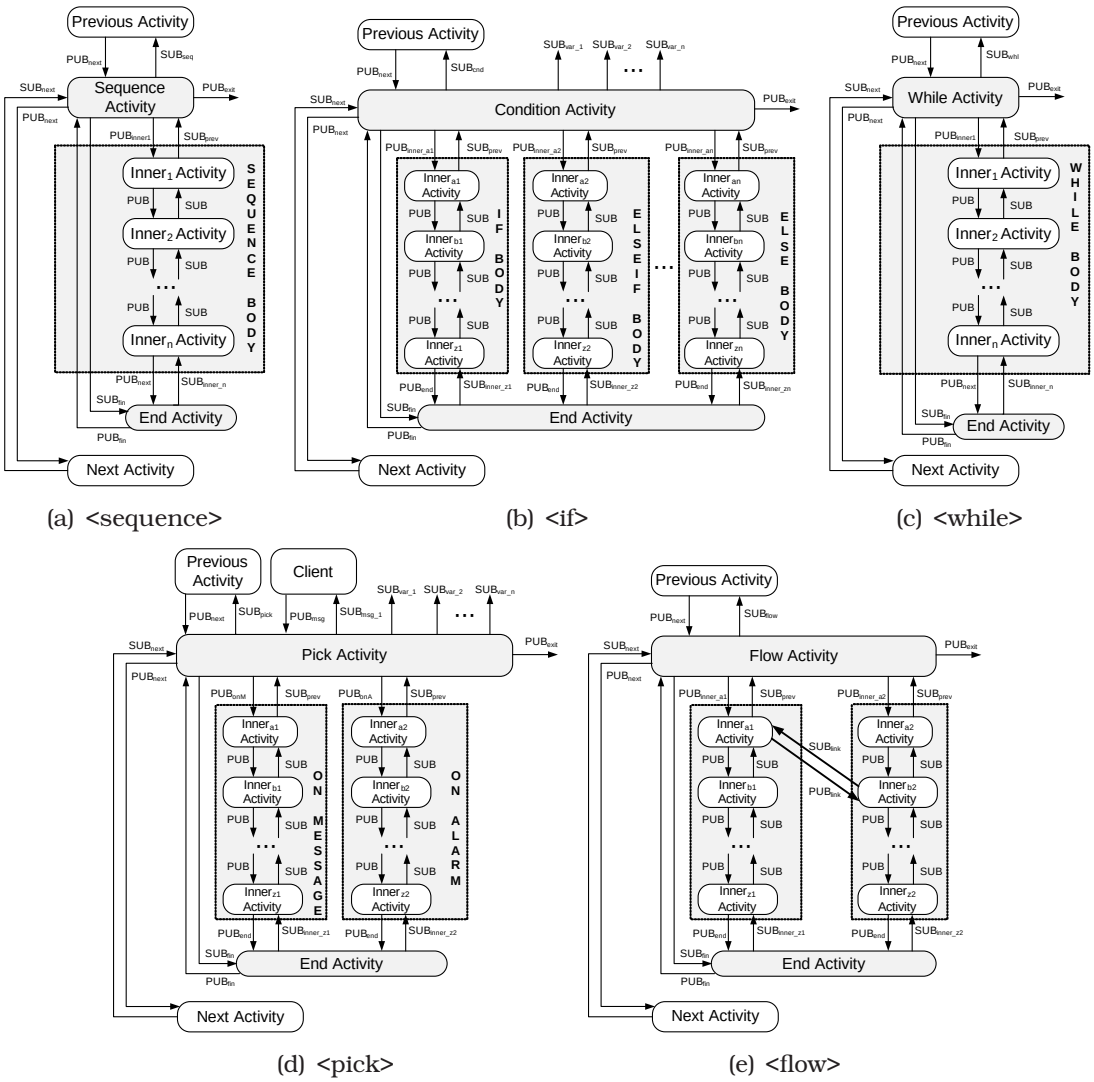
(g) <exit>

(h) <empty>

(i) <wait>

(j) <end> special activity

Figure 4.19: Mapping of the following simple BPEL activities: <receive createInstance=yes>, <receive createInstance=no>, <reply>, asynchronous <invoke>, synchronous <invoke>, <assign>, <exit>, <empty>, <wait>, and <end>[3], into pub/sub language.



(a) <sequence> (b) <if> (c) <while>

(d) <pick> (e) <flow>

Figure 4.20: Mapping of the following structured BPEL activities: <sequence>, <if>, <while>, <pick>, and <flow>, into pub/sub language.

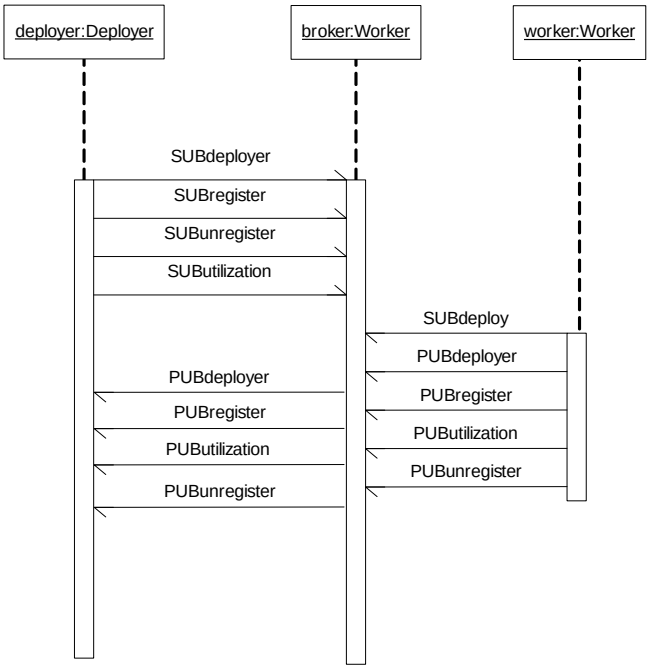


Figure 4.21: Interactions among the deployer, the broker, and the worker nodes during the startup phase.

- *It acts as an entry point.* Thus, any new nodes that join the DHT network and want to participate by becoming worker nodes, register to the deployer node. This is possible because the deployer has a universally known IP address that remains constant during the engine's operation.

- *It acts as a proxy* between the engine and its external clients. To this end, the deployer node receives deployment and execution requests from the BPEL process clients and transforms those requests into appropriate pub/sub messages.

- *It parses the provided* BPEL *process description.* Furthermore, it transforms the process description into pub/sub messages. The latter, are sent to the worker agents during process deployment.

Once the deployer node has started a new DHT network it subscribes using the $\text{SUB}_{deployer}$, $\text{SUB}_{register}$, $\text{SUB}_{unregister}$, and $\text{SUB}_{utilization}$ messages.

Using the $\text{SUB}_{deployer}$ message, the deployer can receive publications from the worker nodes. Thus, when the workers want to directly communicate with the deployer node, they publish a message of the form: $\text{PUB}_{deployer}$=[*class*, DEPLOYER],<<Publication Data>>.

Using the $\text{SUB}_{register}$ and $\text{SUB}_{unregister}$ messages the deployer can receive worker requests for registration or unregistration. These requests are sent as nodes join or leave the DHT network. As the deployer receives the related publication messages, such as

$\text{PUB}_{register}$ and $\text{PUB}_{unregister}$, it can update its Workers Table, which contains information for all the registered workers.

A very important operation of the deployer node is to constantly monitor the worker nodes utilization. Based on this information the deployer determines which nodes will be selected as activity workers during a process deployment. For this reason, the deployer subscribes with the $\text{SUB}_{utilization}$. After the workers join-in, they periodically publish $\text{PUB}_{utilization}$ messages that carry information about their status and utilization. This way, the deployer has accurate information about the nodes state and the network load, and updates the Workers Table with information about all the participating workers. In the $\text{PUB}_{utilization}$ messages, the *stateInfo* attribute contains a normalized meter that describes the utilization of the CPU ($cpu_u(t) \in [0,1]$), the main memory ($mm_u(t) \in [0,1]$), the network ($net_u(t) \in [0,1]$), and the disk ($disk_u(t) \in [0,1]$) of the publishing worker.

Using this information, the deployer calculates the average load on each worker w_i in the Workers Table using the Algorithm 18. When the average load ($\overline{load}_{w_i}$) is over the specified threshold ($l_{threshold}$), the node w_i is considered overloaded.

Algorithm 18: Worker load algorithm

```
for all worker w_i in Workers Table do
    get state info (cpu_u(t), mm_u(t), net_u(t), disk_u(t)) for worker w_i
    get current load_{w_i} for worker w_i
    calculate new load_{w_i} value using equation 4.1.
    if load_{w_i} ≥ l_threshold then
        mark w_i as overloaded node.
    end if
end for
```

Note that in equation (4.1), $w \in [0,1]$ and $\lambda_1 + \lambda_2 + \lambda_3 = 1$. A bigger w value gives more accurate information when the workers use larger intervals to periodically submit their status information, as it emphasizes on the current utilization value. On the contrary, when the publication intervals are very short a lower w is preferred, as it is not influenced by short load bursts. Furthermore, using λ_1, we can give far more importance in computational intensive resources like the CPU and the main memory, as these are more likely to be affected by unfair load balancing.

$$\overline{load}_{w_i}(t) = (1-w) \times \overline{load}_{w_i}(t-1) + w \times (\lambda_1 \cdot (cpu_u(t) + mm_u(t)) + \lambda_2 \cdot net_u(t) + \lambda_3 \cdot disk_u(t)) \tag{4.1}$$

Multiple Deployer Nodes

The previously described scheme, requires some minor modifications when we have multiple deployer nodes. First of all, it is required that all deployer nodes enter the

network before the worker nodes join-in[5]. This restriction applies because we want all the deployer nodes to have the same state by receiving the same publication messages from the workers. As described above, only one deployer node acts as a bootstrap node, but this time this node is elected among the deployer nodes using a leader selection algorithm [HX07, HH06]. This node becomes the entry point for the worker nodes that join the DHT network. After the deployer and the worker nodes join the DHT network they use the same subscription and publication messages as before, but this time the $\text{PUB}_{register}$ and $\text{PUB}_{unregister}$ messages are received by all the deployer nodes. The latter use the received information to construct the same Worker Tables.

At this stage, our engine is ready to receive deployment requests from the external clients. As it uses multiple deployer nodes our engine can receive and handle multiple deployment requests and each request can be served by a separate deployer node. Thus, in this case our engine can provide good load balancing. As each deployer is responsible for deploying a different process, the deployment mechanism becomes fully distributed and exploits its parallelization as this time there is no centralized deployer that becomes a scalability bottleneck.

On the other hand, as the deployer nodes serve separate requests they store data about their deployed processes, that are different among the deployers. Thus, it becomes critical that the deployers replicate their cache data in other deployer nodes and in case of a node failure, a different deployer can take its place or restore the lost state. The cache replication and state restoration problems are not handled in our work, but there are many proposed solutions that can be applied to solve them [LCC+02, CS02]. In order to provide adequate fault tolerance, we decided to represent each deployer by a cluster of machines that maintain the same replicated information. This way, when a deployer failure happens a machine from the cluster can take its position.

Startup Phase: Deployer	
Subscription	**Predicates**
$\text{SUB}_{deployer}$	[*class*, eq, DEPLOYER]
$\text{SUB}_{register}$	[*class*, eq, REGISTER-NODE]
$\text{SUB}_{unregister}$	[*class*, eq, UNREGISTER-NODE]
$\text{SUB}_{utilization}$	[*class*, eq, NODE-UTILIZATION]
$\text{SUB}_{failure}$	[*class*, eq, FAILURE]

Table 4.2: Startup Phase: Deployer node subscription messages.

[5]This is not a hard requirement by our protocol. We could allow the deployer nodes to join at any time into the network, but in this case they must synchronize their state using proper techniques.

Worker Nodes

Each worker is a DHT node with a unique id that is capable of executing the supported BPEL activities. The worker nodes have limited resources and may fail or leave from the DHT network at any time. Every worker node can play a twofold role:

- A worker can act as an *intermediate* pub/sub *broker*, that stores, matches, and publishes pub/sub messages from/to other nodes.
- A worker can act as an *activity executor*, that matches received publication with personal subscriptions, becomes triggered, and executes its deployed activities.

As previously stated, the worker nodes use $\text{PUB}_{register}$ and $\text{PUB}_{unregister}$ messages to register/unregister with their node id to the deployer node. Furthermore, they use $\text{PUB}_{utilization}$ messages to periodically send their utilization status. The latter also carries a timestamp that marks the last time this node executed a process activity. The time interval for the periodic publication is set globally during the DHT startup phase.

After the worker nodes join into the DHT network, they subscribe to the deployer using SUB_{deploy} messages. This way, they can be triggered if the deployer wants to deploy to them a BPEL activity. Additionally, they subscribe using the $\text{SUB}_{undeploy}$ messages. Thus, when a process or instance is undeployed, the related workers are triggered to release all the bound resources by this process instances and to remove all the related information from their cache. Both the SUB_{deploy} and $\text{SUB}_{undeploy}$ messages use the worker's node id. The latter is known to the deployer node through the previously sent $\text{PUB}_{register}$ messages.

Startup Phase: Worker	
Subscription	**Predicates**
SUB_{deploy}	[*class*, eq, DEPLOY-NODE], [*nodeID*, eq,"nodeId"]
Publication	**Attributes**
$\text{PUB}_{deployer}$	[*class*, DEPLOYER], <<nodeInfo>>
$\text{PUB}_{register}$	[*class*, REGISTER-NODE], <<nodeId>>
$\text{PUB}_{unregister}$	[*class*, UNREGISTER-NODE], <<nodeId>>
$\text{PUB}_{utilization}$	[*class*, NODE-UTILIZATION], <<stateInfo, timestamp-last-used>>

Table 4.3: Startup Phase: Worker node subscription and publication messages.

4.4.2 Deployment Phase

In the deployment phase, the deployer is ready to assign BPEL processes to the worker nodes. During this phase, external clients can send process deployment requests to the deployer. When the deployer receives such a request, it parses the BPEL process

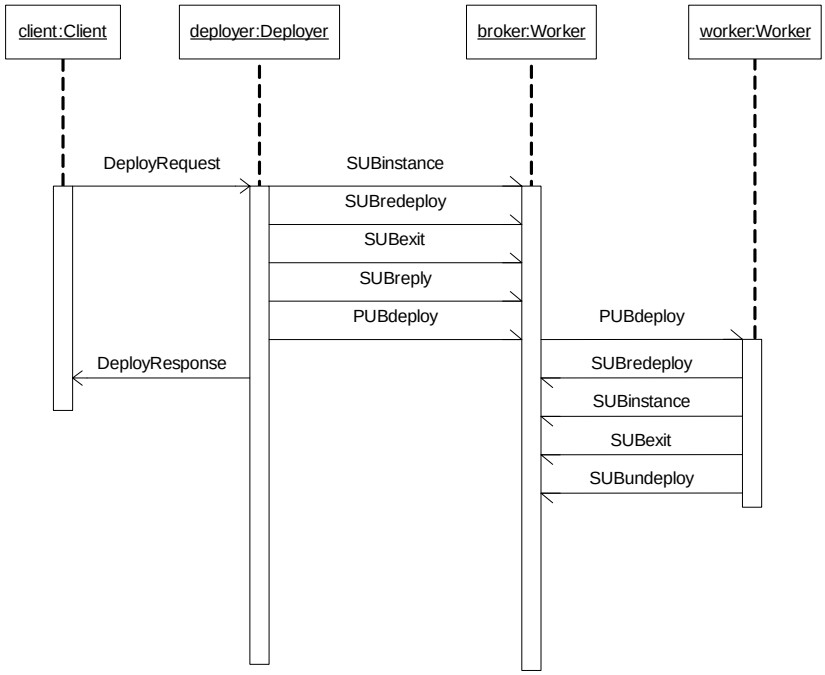


Figure 4.22: Interactions among the deployer, the broker, and the worker nodes during the deployment phase.

and creates a list with the process activities control and data flow dependencies. Then, the deployer assigns the activities to worker nodes with low utilization. Our engine implements two deployment methods:

- *Per-process deployment*. In this method, the deployer selects a set of nodes to deploy the process instances that remain the same for every process instance, i.e. the same worker nodes will execute the same activities for each instance. This method has the benefit of providing low overhead in terms of time and messages during the instance execution, but may create a cluster of overloaded nodes.

- *Per-instance deployment*. In this method, the deployer selects a different set of nodes for each process instance. This method is complementary to the first, as it exhibits good load balancing among the worker nodes, but requires more time and message exchanges to execute a process instance.

Table 4.4 presents the deployer messages, while Table 4.5 presents the worker messages. Figure 4.22 presents all the messages exchanged during the deployment phase between the deployer, the broker, and the worker nodes.

Deployer Node

All the steps that are executed by the deployer during the deployment of a BPEL process are shown in Figures 4.23 and 4.24.

The deployer node becomes triggered when it receives a deployment request from an external client. This request contains a zip bundle with the BPEL file and the associated WSDL files. The deployer unzips the bundle, checks its integrity, and then validates the BPEL/WSDL files. If the check fails, the deployment request is rejected and the client receives a failure reply. Otherwise, the deployment process continues and the deployer parses the BPEL file and constructs a process activity list as the one shown in Figure 4.4(a), while it registers the client's address in the Clients Table.

Then the deployer can use either the per-process deployment or the per-instance deployment to deploy the BPEL process. This decision is made globally during the engine's initialization. In this step the deployer constructs a Process-Activity-Map that associates a process instance with a list of activityData objects that describe process activities and a worker that will execute this activity. Each activityData object contains all the needed information about a single BPEL activity, so the latter can be deployed and executed successfully by the engine, such as:

- the activity id
- the activity XML description
- the publication and subscriptions messages that must be assigned to the worker responsible for executing the activity
- the activity ids of the parent and child activities.

Then the activityData objects along with other pub/sub messages are sent to the selected workers. After this phase completes, the selected workers will have all the needed information for executing the deployed activities. Then using only the deployed pub/-sub messages they start to execute the process instance activities.

Another important role of the deployer is its ability to re-adjust the BPEL process activities deployment during the process execution. This is crucial as the worker nodes can become overloaded or leave at any time. In this case, the failed node activity must be re-deployed using another worker node. Thus, after the process deployment phase is finished, for handling these events the deployer is subscribed using the $\text{SUB}_{redeploy}$ message.

Finally, the deployer node is responsible for handling the successful or failed termination of the process instances. For this reason, the deployer subscribes to these events using the SUB_{exit} subscription. In the following sections we give a more detailed description of the two deployment methods.

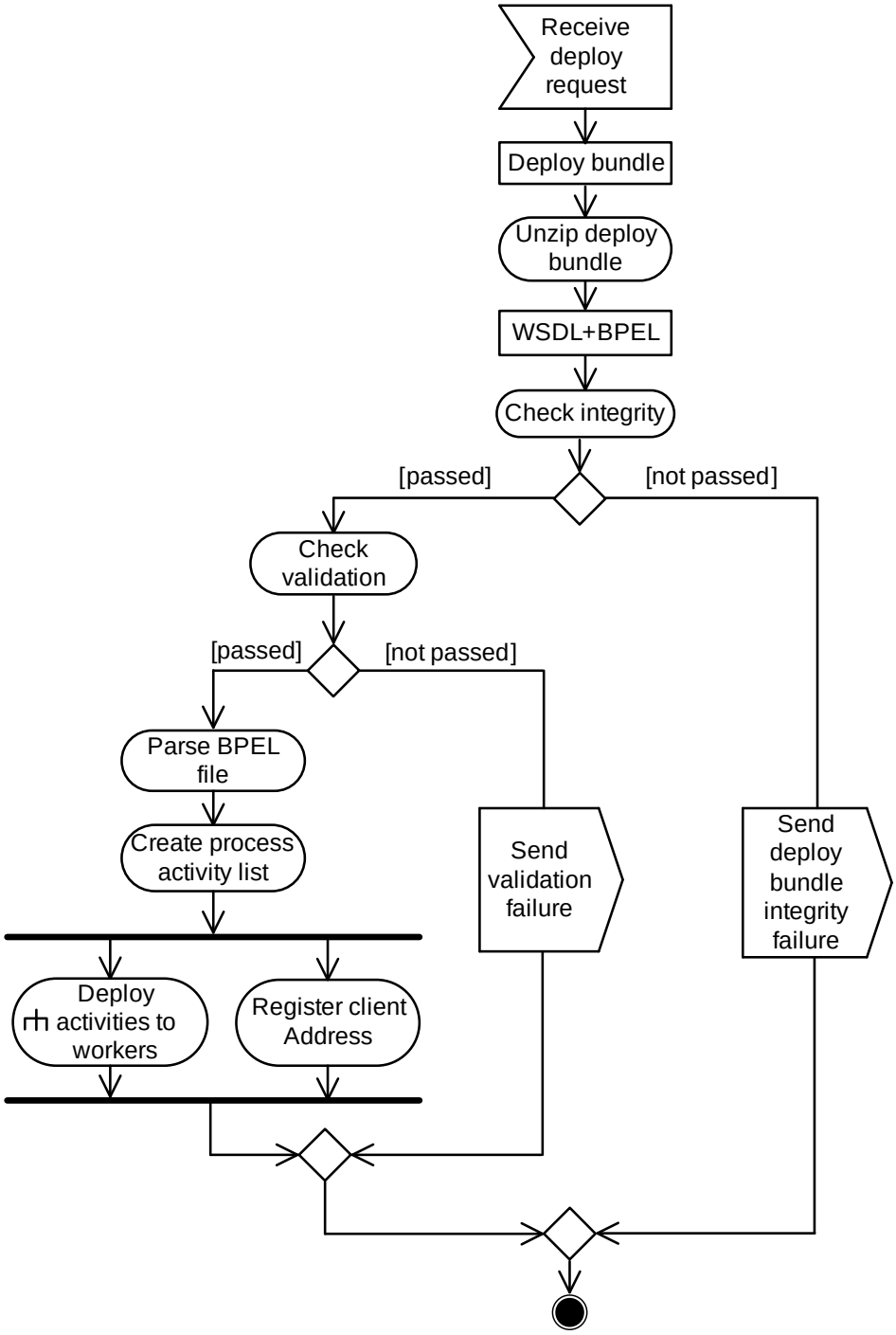
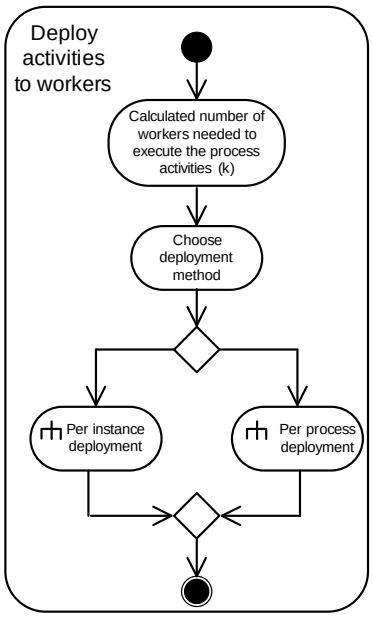
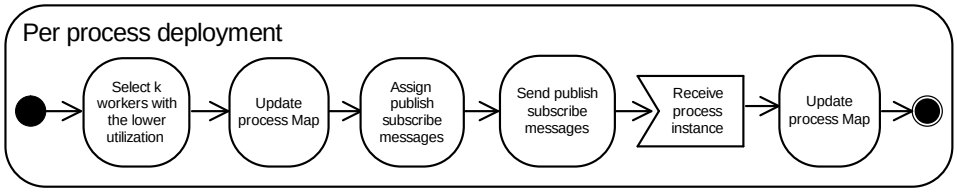


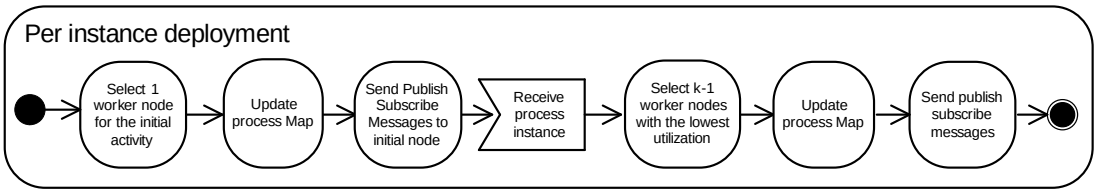
Figure 4.23: Deployment process steps. The deployer receives from the client a deployment request that contains a deploy bundle unzips it, and checks its validity. Then, the deployer parses the BPEL file and creates the activity list that is passed to the selected deployment method. The deployer also registers the client's IP address for future interactions.



(a) Deployment process selection: selects per-instance or per-process deployment



(b) Deployment process: per-process deployment



(c) Deployment process: per-instance deployment

Figure 4.24: The deployer selects a deployment method (Figure 4.24(a)) and then follows the required steps. In per-process deployment, we have only one phase were the deployer assigns the activities to worker nodes. In per-instance deployment we have two phases: a) an initial phase for only the first activity and b) a finalization phase where the deployer selects worker nodes for the remaining activities.

Per-process deployment As we already described, the deployer uses the Algorithm 18 to create a list of non-overloaded worker nodes. This information is used by the per-process deployment (Algorithm 19), along with the timestamp information from each received $\text{PUB}_{utilization}$. This algorithm applies the Least Recently Used (LRU) method and creates a queue of the least recently used, non-overloaded nodes (*workers-LRU-Queue*). These nodes are the ones that will be selected for executing the process activities. For the rest of this section, we assume that the process consists of k activities.

Algorithm 19: Per-Process Deployment

Require: activity-List : list with all process activities
  workers-LRU-Queue ← sort all non overloaded worker's ids using LRU
  for all $\text{activity}_i \in$ activity-List **do**
    worker_{id} ← pop(workers-LRU-Queue)
    assign worker_{id} to activity_i
    send PUB_{deploy}
  end for
  send $\text{SUB}_{instance}$

The deployer chooses a set of k worker nodes[6] from the *workers-LRU-Queue*. Afterwards, the deployer associates each selected worker with an activity in its Process-Activity-List and updates its Process-Activity-Map. The latter associates each process with a number of <activityData, nodeId> pairs. Using these pairs and a number of PUB_{deploy} publications, the deployer assigns to each selected worker the corresponding BPEL process activity. The PUB_{deploy} publications contain the name of the process that will be deployed, the activity type that must be executed, the subscriptions to the preceding activities, and publications to succeeding activities of the process.

Using this deployment method, when the starting worker will be triggered in the execution phase (that is the worker that handles the <receive createInstance=yes> or <pick createInstance=yes> activity), a new process instance will be created. In this case, the workers have already all the needed information in order to send their subscription and publication messages and continue the execution of the process instance. Therefore, using this method the process instance execution achieves a faster startup. On the other hand, as the same nodes are selected for every BPEL process instance, when a process has too many instances these nodes can become overloaded. Thus, the network may exhibit hot spots with overloaded nodes.

Per-instance deployment Similarly, with the previous method the deployer creates a queue of the least recently used, non-overloaded nodes. The per-instance deployment is presented by Algorithm 20. In the description that follows we assume that the process consists of k activities.

[6]This is not always true, in most cases we will need greater than k worker nodes, as for the execution of some structured BPEL activities, we must also use some <end> activities (see appendix A).

Algorithm 20: Per-Instance Deployment

Require: activity-List : list with all process activities
workers-LRU-Queue ← sort all non overloaded worker's ids using LRU
init-activity ← activity with createInstance=yes from activity-List
$worker_{id}$ ← pop(workers-LRU-Queue)
assign $worker_{id}$ to init-activity
send PUB_{deploy} for init-activity
send $SUB_{instance}$
for all $activity_i \in$ {activity-List - initActivity} **do**
  $worker_{id}$ ← pop(workers-LRU-Queue)
  assign $worker_{id}$ to $activity_i$
  send PUB_{deploy} for $activity_i$
end for

The deployer selects only the starting worker (or initial worker) for executing the initial process activity (the activity with <receive createInstance=yes> or <pick createInstance=yes> activity). As soon as it has been selected, a deployment publication PUB_{deploy} is sent to the initial worker. This worker remains the same for all the new instances of the deployed BPEL process as long as it is not overloaded. If the initial worker node becomes overloaded, then a new initial worker node is selected according to the redeployment phase that is presented in section 4.4.4. The deployment of the first activity to the initial worker is equivalent to a per-process deployment of a BPEL process that has only a single activity (one with createInstance=yes). On its part, the deployer subscribes to an instance publication $PUB_{instance}$.

When the initial worker executes its activity, a new process instance is created. The process instance id is published by the worker using a $PUB_{instance}$ message[7]. In this case, the deployer is triggered again due to a matching with the personal subscription $SUB_{instance}$. Then the deployer, using the same method as above, selects $k-1$[8] worker nodes, updates its Process-Activity-List and Process-Activity-Map by assigning activities to workers, and publishes the deploy information to the selected $k-1$ workers using PUB_{deploy} messages.

The per-instance deployment is complementary to the previous solution. It has a slower startup for the instance execution, but provides better load-balancing as the deployer can choose for every process instance different workers for the same activity. This happens, because the deployment phase is separated to two steps. In the first step the deployer selects the staring worker (that is the worker that handles the <receive createInstance=yes> or <pick createInstance=yes> activity) and assigns to it the proper subscription and publication messages. When this node is triggered, a new instance id is created and a publication is sent to the depoyer, triggering the second step of the

[7]This message will always have the attribute storePub=yes, so it can be stored into the historic cache and matched by future subscriptions.

[8]We assume that k is the number of the BPEL process activities.

per-instance deployment: the deployer select workers for the rest of the BPEL process activities, assigns to the workers the proper subscription and publication messages, and triggers the next activity in the BPEL process. As the deployer has more accurate node status information during the second step of the per-instance deployment, it can select worker nodes with less load for executing this process instance.

Deployment Phase: Deployer	
Subscription	**Predicates**
$SUB_{instance}$	[*class*, eq, INSTANCE], [*process*, eq, "processName"]
$SUB_{redeploy}$	[*class*, eq, REDEPLOY], [*processName*, eq, "processName"]
SUB_{exit}	[*class*, eq, EXIT], [*process*, eq, "processName"]
SUB_{reply}	[*class*, eq, REPLY], [*process*, eq, "processName"]
Publication	**Attributes**
PUB_{deploy}	[*class*, DEPLOY-NODE], [*nodeID*, "nodeId"], <<processName, activityId, List<Sub>, List<Pub>>>
$PUB_{undeploy}$	[*class*, DEPLOY-NODE], [*nodeID*, "nodeId"], [*process*, eq, "processName"]

Table 4.4: Deployment Phase: Deployer node subscription and publication messages.

Worker Nodes

If a matching PUB_{deploy} message arrives, the worker becomes responsible for executing the associated process activity. Based on the used deployment mechanism, this node will execute the specific activity for every instance of the BPEL process, or will execute this activity only for the specific process instance.

If the activity is a <receive createInstance=yes> or <pick createInstance=yes>, the worker can subscribe and wait for the proper invocation. When triggered, it executes the activity and sends a $PUB_{instance}$ message. The target(s) of the $PUB_{instance}$ depend on the used deployment mechanism. If the engine uses a per-instance deployment, then the $PUB_{instance}$ is sent only to the deployer node. On the other hand, if we use per-process deployment, the $PUB_{instance}$ is propagated to all the worker nodes of the BPEL process.

4.4.3 Execution Phase

During the execution phase the process instance activities execution takes place. Table 4.6 presents the deployer messages, while Table 4.7 presents the worker messages.

In the start of this phase the worker responsible for the initial process activity is triggered by the deployer. This node executes its assigned activity and sends two types of publication messages: a) publication message to the next activity and b) publication message to the deployer and worker nodes that carry the process instance id.

Deployment Phase: Worker	
Subscription	**Predicates**
$\text{SUB}_{instance}$	[*class*, eq, INSTANCE], [*process*, eq, "processName"]
SUB_{exit}	[*class*, eq, EXIT], [*process*, eq, "processName"]
$\text{SUB}_{undeploy}$	[*class*, eq, UNDEPLOY-NODE], [*nodeID*, eq, "nodeId"], [*process*, eq, "processName"]

Table 4.5: Deployment Phase: Worker node subscription messages.

When the worker nodes responsible for the rest of the process activities are in turn triggered, they execute them and produce new publications. This process ends when the last process activity is successfully executed and the associated worker produces a PUB_{reply} (as shown by Figure 4.25) or PUB_{exit} message with a success description (as shown by Figure 4.26). In case of fault, a publication PUB_{exit} message is sent that carries an error description (as shown Figure 4.27).

Deployer Node

When an external client invokes a deployed BPEL process[9] the deployer is triggered. Then it translates the client's request to a PUB_{msg_i} for an initial <pick createInstance=yes> activity, or to a $\text{PUB}_{rcv_{yes}}$ message for an initial <receive createInstance=yes> activity. The initial BPEL process activity is triggered and executed by its worker node. After the execution is finished, a new process instance id is published by the worker node, using a $\text{PUB}_{instance}$ message.

If the execution of the last BPEL process activity is finished, then the deployer is triggered by a PUB_{exit} or PUB_{reply} message. If the deployer receives a PUB_{reply} or a PUB_{exit} <<SUCCESS>> message, the process instance execution completes successfully. Furthermore, in the first case the data carried by the PUB_{reply} publication are send back to the external client that originally invoked the BPEL process instance. In case the message is PUB_{exit} <<FAILURE>>, the deployer stores to a log the process instance failure details. In any case the deployer removes this process instance from its Process-Instances Table.

Worker Nodes Execution

At some point, the workers that were subscribed to instance publications for the deployed process will receive a $\text{PUB}_{instance}$ message. Then, they will update their Process-

[9]The deployer acts as a proxy between the external clients and the pub/sub overlay. When a client makes a request to execute a process, the deployer translates the request to a publication that is propagated to the worker responsible for the initial activity. There are other alternative solutions to this problem; we could use a server activity as a proxy that communicates with the external clients and transfers the invocation requests to the proper worker node, or the client could directly call the worker node.

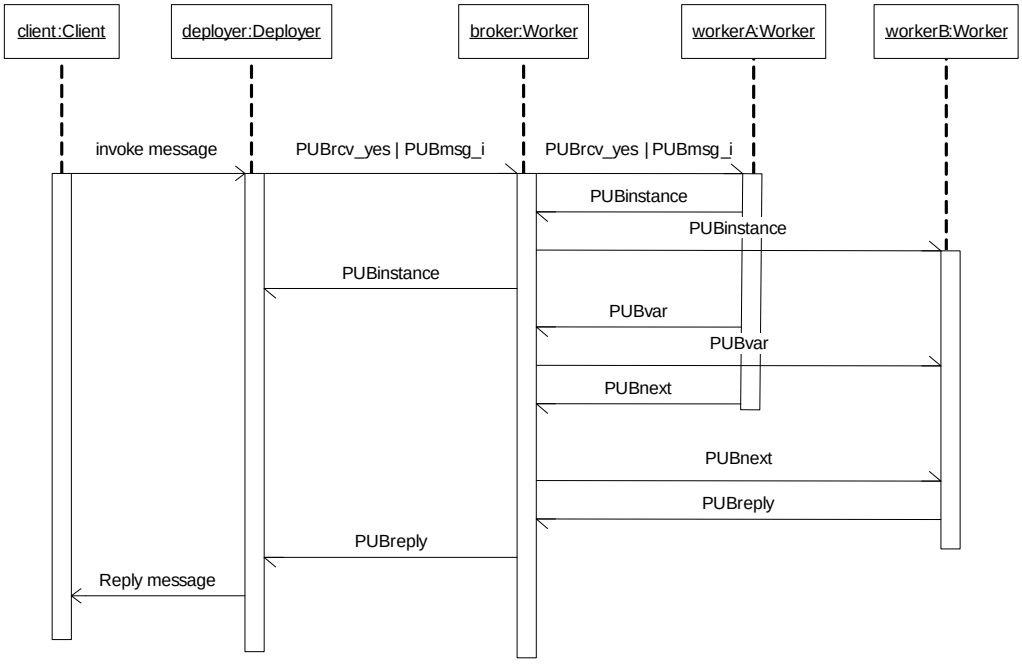


Figure 4.25: Interactions among the deployer, the broker, and the worker nodes during the execution phase. In this case, we have a successful execution that ends with a reply.

Instances Tables. Afterwards, each worker can use the new instance id to update its deployed pub/sub messages: it creates instances of the publications and subscriptions in the List<Pub> and List<Sub> that were received with the PUB_{deploy} messages. Furthermore, each worker creates correlation matching trees to provide matching for composite subscriptions[10]. Finally, the workers send their deployed subscriptions. When publications from other nodes are matched with these subscriptions on the available broker nodes, they will be disseminated towards the original subscriber.

When a worker node receives a publication, it checks for a match using its correlation matching tree. If there is a match, the worker executes the corresponding activity, and publishes the produced publication messages with the updated variables/results using PUB_{var}. The worker also triggers the next activity using PUB_{next} messages. This process continues, until the execution is finished or an error happens.

There is one special case concerning the initial worker node that is responsible for the activity with createInstance=yes. When this worker submits the $\text{PUB}_{instance}$ message, it may reach the broker node before the workers that belong to the same process instance send their own subscriptions about it ($\text{SUB}_{instance}$). This causes a problem, as the pub/sub matching will fail and the execution of the process will stall. For this reason, the initial worker sets a flag (storePub=yes) into its $\text{PUB}_{instance}$ message. This causes the publication to be stored into a historic cache of the broker. Thus, when a subscription

[10]More information on the used composite subscription is provided in the Appendix A

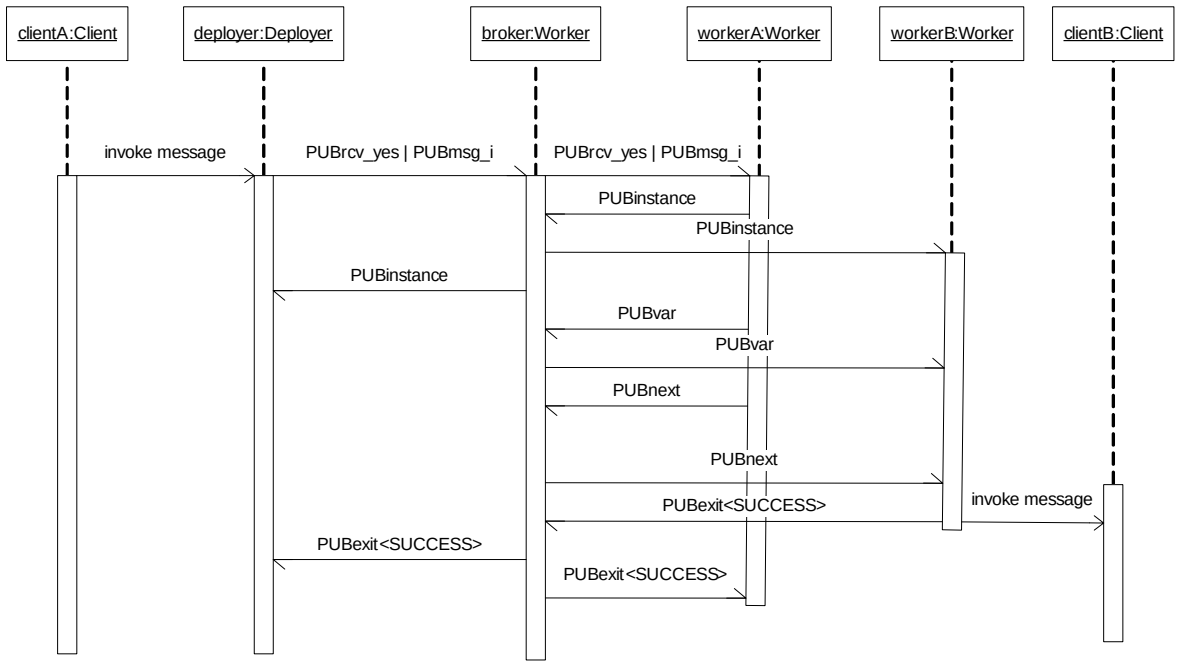


Figure 4.26: Interactions among the deployer, the broker, and the worker nodes during the execution phase. In this case, we have a successful execution that ends with an exit<<SUCCESS>>.

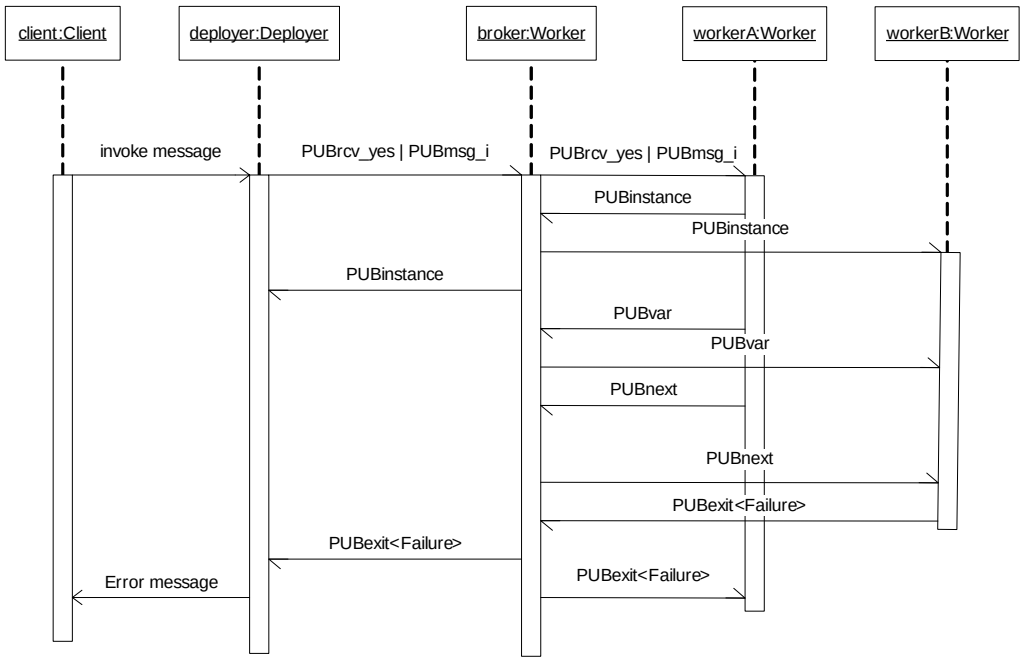


Figure 4.27: Interactions among the deployer, the broker, and the worker nodes during the execution phase. In this case, we have a failed execution that ends with an exit<<FAILURE>>.

Execution Phase: Deployer	
Publication	**Attributes**
$\text{PUB}_{rcv_{yes}}$	[*class*, ACTIVITY], [*process*, "processName"], [*activityID*, "activityId"], [*variableName*, "BPELVariableName"], [*partnerLink*, "NCName"], [*portType*, "QName"]?, [*operation*, "NCName"]
PUB_{msg_i}	[*class*, ACTIVITY], [*class*, ACTIVITY], [*process*, "processName"], [*activityID*, "activityId"], [*variableName*, "BPELVariableName"], [*partnerLink*, "NCName"], [*portType*, "QName"]?, [*operation* "NCName"]

Table 4.6: Execution Phase: Deployer node publication messages.

arrives later to the broker node, the matching is performed into the historic cache using the previously arrived publications.

Execution instance terminated with success In this scenario, the execution reaches an <exit> activity or a <reply> activity. In the first case, a PUB_{exit} <<SUCCESS>> publication is sent, and is propagated to the deployer node as well as to all worker nodes that participate in the process and to the deployer node. When a worker receives a PUB_{exit} <<SUCCESS>> it removes all the subscriptions that match the processName and instance id of the publication from:

- the Personal-Subscription Cache
- the Broker-Subscription Cache
- the Historic Cache
- the Process-Instances Table
- the Activities Table.

In the second case, the worker that executes a <reply> activity sends a PUB_{exit} <<SUCCESS>> message (that is handled as described above) and a PUB_{reply} message. The latter is propagated to the deployer node, which informs the external client with a response carrying the produced results.

Execution instance terminated with failure In this scenario, a failure publication is sent via PUB_{exit} <<FAILURE>>. This message is propagated to all worker nodes that participate in the process and to the deployer node. When a worker receives this publication, it removes all the subscriptions that match the processName and the instance id of the publication from:

- the Personal-Subscription Cache
- the Broker-Subscription Cache
- the Historic Cache
- the Process-Instances Table
- the Activities Table.

Execution Phase: Worker	
Subscription	**Predicates**
SUB_{prev}	subscription to the previous activity
SUB_{var}	[*class*, eq, VARIABLE_UPDATE], [*process*, eq, "processName"], [*instanceID*, eq, "instanceId"], [*activityID*, eq, "activityId"], [*variableName*, eq, "BPELVariableName"]
Publication	**Attributes**
PUB_{exit}	[*class*, EXIT], [*process*, "processName"], <<instanceId, "SUCCESS | FAILURE ">>
$PUB_{instance}$	[*class*, INSTANCE], [*process*, "processName"], <<instanceId>>
PUB_{var}	[*class*, VARIABLE_UPDATE], [*process*, "processName"], [*instanceID*, "instanceId"], [*activityID*, "activityId"], [*variableName*, "BPELVariableName"], <<variableData>>
PUB_{next}	publication to the next activity
PUB_{reply}	[*class*, REPLY], [*process*, "processName"], <<instanceId, BPELVariableName, variableData>>

Table 4.7: Execution Phase: Worker node subscription and publication messages.

4.4.4 Redeployment Phase

During the execution phase the deployer node may need to deploy some process activities to different nodes. This happens because a worker node may become overloaded, may leave the network, or fail. Thus, the process instance execution must stop and continue only after a new node takes the place of the erroneous node during the redeployment phase. Table 4.8 presents the worker messages.

Worker nodes leaves or becomes overloaded

If a worker node leaves the network (or becomes overloaded), it sends a PUB_{exit} <<FAILURE>> message to inform all the related worker nodes and the deployer that the execution will be interrupted. This message is handled as described in the execution

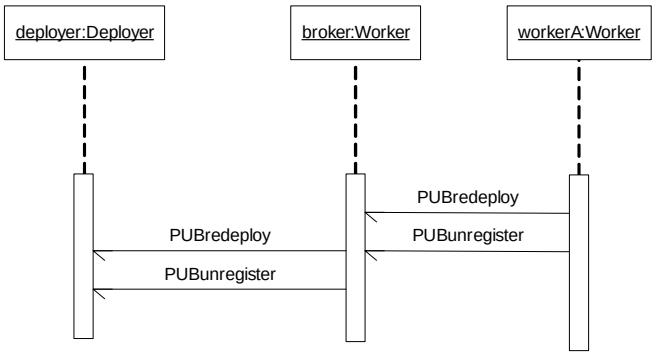


Figure 4.28: Interactions among the deployer, the broker, and the worker nodes during the redeployment phase. In this case, a node leaves the DHT network.

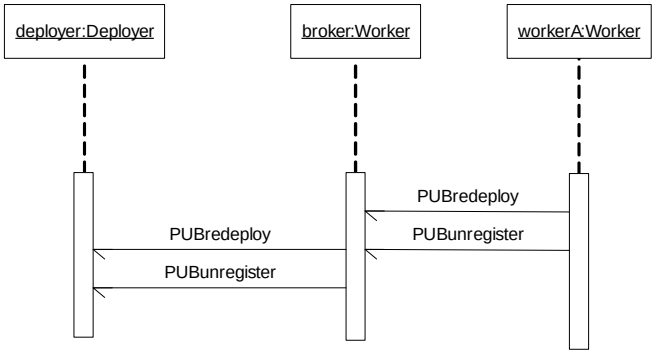


Figure 4.29: Interactions among the deployer, the broker, and the worker nodes during the redeployment phase. In this case, a node fails.

phase. Afterwards, the node that leaves sends a $\text{PUB}_{redeploy}$ message to the deployer node. Finally, the worker leaves by sending a $\text{PUB}_{unregister}$ to the deployer node.

When the deployer receives a $\text{PUB}_{redeploy}$ message, it chooses another node to take the position of the leaving worker node. This is achieved by using the appropriate deployment algorithm to select another worker with low utilization (the node that sent the redeploy message is discarded from the selection algorithm). Then, the deployer updates its Process-Instances Table entries and triggers again the failed activity. Finally, the deployer triggers again the initial process activity to restart the process instance execution. The $\text{PUB}_{unregister}$ message causes the deployer to remove this node from its Workers Table.

Worker node failure

When a worker node discovers that a node in its routing table has failed (in this case the *MSPastry* layer informs the pub/sub layer for a missing routing table entry), it sends a $\text{PUB}_{failure}$ to inform the deployer node for a worker failure. The deployer matches the node id with the associated entry in its Process-Instances and Workers Tables and sends a PUB_{exit} <<FAILURE>> message. This publication informs all the workers that were in the same process/instance with the failed node. Then the deployer starts the redeployment phase and replaces the failed worker with a new one, in all the related process/instances. Finally, the deployer restores the failed instances by triggering again the initial activity in each process instance.

Redeployment Phase: Worker	
Publication	**Attributes**
$\text{PUB}_{redeploy}$	[*class*, REDEPLOY], [*processName*, eq, "processName"], <<nodeId, instanceId>>
$\text{PUB}_{unregister}$	[*class*, UNREGISTER-NODE], <<nodeId>>
$\text{PUB}_{failure}$	[*class*, FAILURE], <<nodeId>>

Table 4.8: Redeployment Phase: Worker node publication messages.

4.4.5 Undeployment Phase

In the undeployment phase, the deployer receives a request from an external client to remove a BPEL process and its associated instances. Thus, any related process instance must stop and the related resources must be released. Table 4.9 presents the worker messages.

In this case, the deployer removes from its Process-Instances Table the entry with the particular process name and sends a $\text{PUB}_{undeploy}$ message. The latter, is handled by all worker nodes that participate in any instance of the particular process. Thus, the

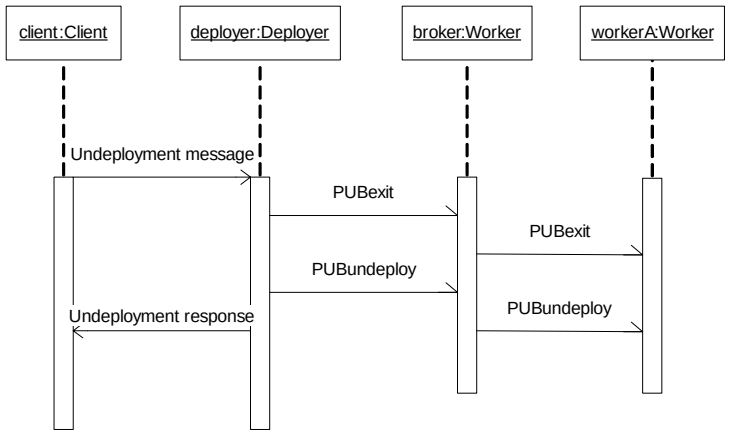


Figure 4.30: Interactions among the deployer, the broker, and the worker nodes during the undeployment phase.

workers remove any subscription that is associated with the particular process entry from their:

- Process-Instances Table
- Personal-Subscription Cache
- Correlation Cache.

Furthermore, the deployer sends a PUB$_{exit}$ <<FAILURE>> message to every instance of the undeployed process. This way, the worker nodes that acted as brokers by storing the associated subscriptions will remove them from their Broker-Subscription Cache.

Undeployment Phase: Worker	
Publication	**Attributes**
PUB$_{exit}$	[*class*, EXIT], [*process*, "processName"], <<instanceId=ANY, FAILURE>>
PUB$_{undeploy}$	[*class*, UNDEPLOY-NODE], [*nodeID*,"nodeId"], [*process*, "processName"]

Table 4.9: Undeployment Phase: Worker node publication messages.

4.5 Conclusions

In this chapter, we presented in full detail the design and architecture of our proposed orchestration engine, called ADORE. We focused mainly on three central aspects of our engine:

1. We presented the algorithms used by our pub/sub layer and we analyzed their cost and efficiency. We covered our proposed algorithms for pub/sub matching, that exploit subscription covering and merging techniques. As we selected an infrastructureless approach based on a DHT network and we implemented our engine on top of a provided pub/sub mechanism, it becomes clear that its efficiency is critical for the engine's operation. Our engine uses three algorithms for the subscription storage namely:

   (a) the Random-Predicate Subscription Algorithm (RP-SA)

   (b) the Proximity-Predicate Subscription Algorithm (PP-SA)

   (c) the Multi-Predicate Subscription Algorithm (MP-SA).

   Furthermore, it uses two algorithms for sending publications:

   (a) the Single-Attribute Publication Algorithm (SA-PA)

   (b) the Multi-Attribute Publication Algorithm (MA-PA).

2. We presented the translation of the BPEL constructs in pub/sub messages that can be used by the pub/sub layer. We gave information for the translation of simple and structured BPEL activities.

3. We presented the operation our engine during its five main phases:

   - startup
   - deployment
   - execution
   - redeployment
   - undeployment.

   We analyzed how the deployer, broker, and the worker nodes operate in each phase. Furthermore, we gave an overview about the format of the used pub/sub messages in each phase. Finally, we illustrated all message interactions that happen when each phase completes successfully or when an error happens.

Chapter 5

Evaluation

This chapter discusses the evaluation methodology followed in this thesis and comments on its results. Section 5.1 presents the *PeerSim* simulation engine that was used to implement our proposed architecture and articulates on both the benefits and drawbacks of its use. Furthermore, we rationalize our decision to use *PeerSim* as a testbed for evaluating our work. In section 5.2 we present simulation results from the evaluation of the implemented pub/sub mechanisms. The latter are crucial for the engine's correct and efficient operation. We demonstrate that our pub/sub algorithms scale well and exhibit good load balancing features. Then, in section 5.3 we evaluate the engine's operation, by deploying and executing multiple process instances under varying conditions. We measure the engine's performance versus a centralized clustered engine in terms of the processes average execution time and throughput. Furthermore, we compare the performance of the two proposed deployment mechanisms (i.e per-process or per-instance deployment). Finally, in section 5.4 we present the concluding remarks.

5.1 Simulation

We decided to implement our architecture on the *PeerSim*[1] simulator for three main reasons:

1. It can provide realistic (simulating transport layer delays) large-scale simulations.

2. It can be easily extended with new protocols (i.e protocols that modify the nodes state, change the network nodes churn rate, etc.). This way, we can create simulations for different conditions, as the user can change a number of simulation parameters and experiment with the implementation.

3. It allows the user to run experiments using the exact same conditions (i.e network delays, node links, etc). Therefore, we can be sure that the results from the

[1] http://peersim.sourceforge.net/

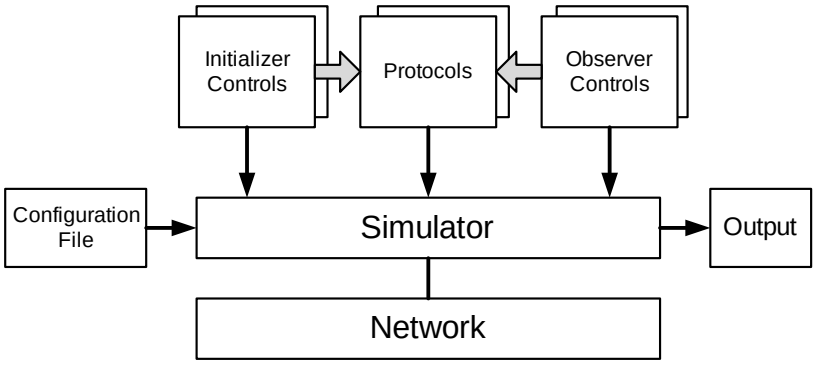


Figure 5.1: PeerSim simulator

experiments are derived from our protocols behavior and not due to some artifact or a unique network condition. In short, using the simulator we can examine the correctness of our implementation.

We implemented ADORE on top of *PeerSim* [MJ09] simulator. *PeerSim* is a Java framework designed for experimentation with large scale P2P overlay networks. *PeerSim* started under EU project BISON[2] [MB02, BCD+05]. Then was used by the EU project DELIS[3] [del06] and is now partially supported by the Napa-Wine[4] [RB11, CdSLMM11] EU project.

PeerSim has been developed with extreme scalability in mind and support for network dynamicity. It is released to the public under the GPL open source licence[5] and consists of two different simulation engines: a *cycle-based engine* and an *event-based engine*. The cycle-based engine allows the pursuit of maximum scalability and uses some simplifying assumptions, such as ignoring the details of the transport layer in the communication protocol stack. On the other hand, the event-based engine is less efficient but more realistic, as it supports transport layer simulations. Thus, using the event-based engine, the *PeerSim* can model both random delays and message drops. Furthermore, the event-based engine uses an internal representation of time[6] to provide timing information about the execution delay. This timing meter is zero at startup and it is advanced by message delays until the user-defined end time (that marks the end of the simulation). Thus, the simulation stops when the event queue is empty (nothing left to do), or if all the events in the queue are scheduled for a time later than the specified end time.

Both cycle and event-based engines are supported by the use of many simple, extensible, and pluggable components that are matched by a flexible configuration mechanism. The *PeerSim* architecture is presented in Figure 5.1. The *PeerSim* simulator

[2] http://www.cs.unibo.it/bison
[3] http://delis.upb.de/
[4] http://napa-wine.eu/
[5] http://www.gnu.org/copyleft
[6] this is a long value (64 bit integer)

was designed to encourage modular programming. Therefore, it is based on extensible building blocks. Moreover, every block can be easily replaced by an another component that implements the same interface (i.e. provides the same functionality). In general the simulation model is used according to the following steps:

1. Choose a network size (number of nodes).
2. Choose one or more *Protocol* objects to experiment with and initialize them. These are specialized protocols implemented by the user.
3. Choose one or more *Control* objects to monitor the properties you are interested in, or to modify some of the simulation parameters (e.g the size of the network, the internal state of the nodes, etc).
4. Run the simulator invoking an object of the *Simulator* class, using a configuration file that contains the initial simulation parameters.

We implemented our architecture on top of the event-based engine. For that reason, we proceed with a short description of the event-based engine's life-cycle operation.

The first step for the simulation's execution is to read the configuration file, given as a command-line parameter. The configuration file contains the simulation parameters concerning all the objects involved in the experiment. Then, the simulator sets up the network, initializes the network nodes, and instantiates their protocols. Each node has the same kinds of protocols. Thus, instances of a protocol form an array in the network with one instance in each node. The instances of the nodes and the protocols are created by cloning. That is, only one instance is constructed using the constructor of the object, which serves as prototype, and all the nodes in the network are cloned from this prototype. At this point, initialization needs to be performed, that sets up the initial states of each protocol. The initialization phase is carried out by *Control* objects (*Initializers*), that are scheduled to run only at the beginning of each experiment. After initialization, the event driven engine calls the components (*Protocols* and *Controls*) based on each message arrival time in event queue. This process continues until the queue is empty, or the end of the simulation is reached. Finally, specialized *Control* objects are called that collect data (*Observers*). The latter format the data and send them to the standard output. This way, the user can easily redirect the simulation output to a file and collect it for further work.

5.2 ADORE Publish/Subscribe Evaluation

The main objective of our evaluation was to assess the scalability of the orchestration engine through the use of a DHT overlay network and an efficient pub/sub mechanism. Thus, we first evaluated the proposed pub/sub mechanisms[7] (*RP-SA*, *PP-SA*, *MP-SA*). In this context, we pursued four specific goals:

[7]These methods were originally presented in section 4.2.

1. *Pub/Sub correctness*. The pub/sub mechanisms must match subscription with publication messages and disseminate the matched publication messages to all the subscriber nodes.

2. *Pub/Sub efficiency*. The pub/sub mechanisms must require minimum number of hops for the publication messages to reach a subscriber node. Furthermore, they must not overload the broker nodes.

3. *Pub/Sub effectiveness*. The pub/sub mechanisms must provide publication matching and dissemination, even when the average number of subscribers increases, by exhibiting similar overload with the typical case.

4. *Pub/Sub scalability*. Even in large networks, with increased number of publication and subscription messages, the average number of required hops and the nodes overhead must not increase significantly.

5.2.1 Metrics

We used a set of metrics to evaluate the performance and cost of our proposed pub/sub mechanisms:

1. *Hops*. The average number of overlay hops taken to deliver an event to all of its subscribers.

2. *Latency*. The average time taken to deliver an event to all of its subscribers.

3. *Overhead*. The ratio of the number of intermediate nodes involved during the delivery of an event to the number of subscribers for this event. The lower the overhead, the better the performance.

4. *Bandwidth Cost*. The ratio of the total bandwidth cost incurred by an event delivery to the number of nodes involved.

5.2.2 Setup

To run our experiments we used a network consisting of 1,024 nodes, with average connectivity degree k=5, and transmission latencies defined between 100 ms and 500 ms. Each node used a 128-bit identifier and the parameter for *MSPastry* was b=4. This means that each node had a routing table R with 3 rows and 15 columns, a neighbor set M with 32 nodes, and a L leaf set with 16 nodes[8].

In our simulation, new nodes were joining into the system until the total number of nodes was reached (e.g. 1,024 nodes). After the system stabilization, we started the subscription installation that caused the system to register nodes as subscribers. When the subscription installation was over, the event publication phase began. The latter

[8]These numbers are produced using the Pastry equations presented in section 2.5.2.

was modeled using the Poison distribution with a mean arrival rate of 50 publications per minute. We used the pub/sub scheme that was proposed by Meghdoot [GSAA04]:

S = [Date:string, 2/Jan/98, 31/Dec/02],
[Symbol:string, "aaa", "zzz"],
[Close:float, 0, 500],
[High:float, 0, 500],
[Low:float, 0, 500],
[Volume:integer, 0, 310,000,000]

Specifically, *Symbol* is the stock name. *Close* is the closing price for a stock a given day. *High* and *Low* are the highest and lowest prices for the stock on that day. *Volume* is the total amount of trade in the stock on that day.

We generated subscriptions using the five subscription templates suggested in Meghdoot:

$T_1 = [(\text{Symbol} = P_1) \wedge (P_2 \leq \text{Close} \leq P_3)]$ with probability 20 percent,

$T_2 = [(\text{Symbol} = P_1) \wedge (\text{Low} \leq P_2)]$ with probability 35 percent,

$T_3 = [(\text{Symbol} = P_1) \wedge (\text{High} \geq P_2)]$ with probability 35 percent,

$T_4 = [(\text{Symbol} = P_1) \wedge (\text{Volume} \geq P_1)]$ with probability 5 percent, and

$T_5 = [(\text{Volume} \geq P_1)]$ with probability 5 percent

The logic behind this distribution is that templates with more general interests (e.g. T_4 and T_5) are assigned low probabilities. This is based on the fact that in a real application, the subscribers are usually interested in specific events related to their narrow interests [GSAA04], rather than in more general information.

We selected the Meghdoot pub/sub scheme and templates for three reasons. The first one, is that its templates have become a common base for comparing the efficiency of the proposed pub/sub mechanisms. Second, it offered us the flexibility to evaluate our implementation with different distributions of subscription and publication messages. Finally, this scheme is directly related to the subscription templates that are used by our BPEL engine (as presented in the Appendix A). Templates T_4 and T_5 are similar to the deployer subscriptions $\text{SUB}_{register}$, $\text{SUB}_{unregister}$, $\text{SUB}_{deployer}$, $\text{SUB}_{redeploy}$, $\text{SUB}_{utilization}$, and $\text{SUB}_{failure}$. Moreover, the templates T_1, T_2, and T_3 act as worst case scenario for the worker nodes subscriptions (in most cases only a single worker subscription is matched with a publication message).

5.2.3 Experimental Results

In this section, we first evaluate the performance of our pub/sub mechanisms under the standard configuration that was presented above. Then, we examine their performance using different ranges of subscribers and network sizes.

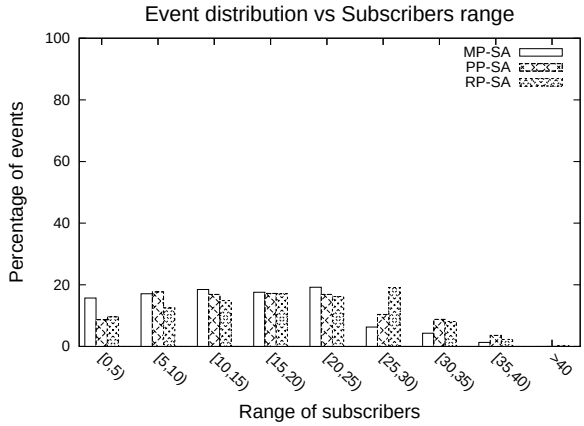


Figure 5.2: Average number of subscribers per event.

5.2.3.1 Performance under Standard Configuration

Under standard configuration we performed simulations where we generated randomly 100,000 events. The average number of subscribers per-event, produced by our subscription storage algorithms (*RP-SA, PP-SA, MP-SA*) is presented in Figure 5.2. As illustrated, the number of average subscribers per-event is 17, or approximately about 1.7% percent of the 1,024 nodes.

Table 5.1, presents the performance of the different storage algorithms. The *PP-SA* and *MP-SA* outperform the *RP-SA* in the required number of hops, latency, bandwidth, and overhead. For the *PP-SA*, this happens because the event delivery messages traverse over shorter paths, as the broker responsible for the storage and matching is closer to the subscriber node. The *MP-SA* exploits another mechanism to exhibit better performance: as most of the subscriptions are stored to a small subset of broker nodes, the group delivery mechanism can easily group many publications with common delivery path using only one message. Thus, by exploiting the common delivery paths it reduces the average number of hops required for the event delivery. This causes the *MP-SA* algorithm to exhibit the best bandwidth cost between the three.

Scheme	**hops**	**latency (ms)**	**bandwidth cost (bytes/node)**	**overhead**
RP-SA	3.512	334.506	171.996	2.421
MP-SA	2.793	225.32	**147.244**	1.738
PP-SA	**2.717**	**202.586**	156.904	**1.524**

Table 5.1: Performance comparison between the RP-SA, MP-SA, and PP-SA subscription algorithms.

Figure 5.3(a) plots the distribution of events for *PP-SA*, *MP-SA*, and *RP-SA* according to the range of hops. As it is shown, all events require 2 to 5 hops to reach the subscribed

nodes. For *PP-SA* and *MP-SA* algorithms, almost 90 percent of the events are delivered to their subscribers within two to three hops. The average number of delivery hops is 2.717 for the *PP-SA* and 2.793 for the *MP-SA*.

Figure 5.3(b) plots the distribution of events for *PP-SA*, *MP-SA*, and *RP-SA*, according to the range of latency. The majority of the events is delivered between 100 and 350 ms. The *PP-SA* in this case is the winner, as its proximity metrics store the subscriptions to nodes that are closer to the subscribers. Thus, when the events are matched they need to travel smaller distances to reach the original subscribers. Furthermore, these links exhibit low latency due to their network proximity. We notice that in the case of the *MP-SA* there is a spike of events with delivery latency around 200 and 250 ms. This is again a product of the group message delivery. The simulation engine does not create delays based on the message sizes, but randomly according to the links and their network proximity. Thus, the group delivery mechanism using *MP-SA* does not pay a toll for sending larger messages, but takes advantage of the reduced delivery path. The *RP-SA* mechanism exhibits larger delivery latencies as it may send a publication to a broker node that is on the other side of the DHT ring. Thus, the message requires to travel through additional nodes.

Figure 5.3(c) plots the distribution of events for *PP-SA*, *MP-SA*, and *RP-SA* according to the range of bandwidth. This time the winner is the *MP-SA* as it groups the events in a smaller number of messages, thus requiring less bandwidth per subscriber.

Figure 5.3(d) plots the distribution of events for *PP-SA*, *MP-SA*, and *RP-SA* according to the range of overhead. Clearly, all algorithms produce good results as the majority of the events produces low overhead, thus requiring few intermediate nodes.

Figure 5.4 shows subscription distribution over five broker nodes for the *PP-SA*, *MP-SA*, and *RP-SA* algorithms. Note that *RP-SA* evenly distributes subscriptions to the broker nodes, as it chooses randomly a subscription predicate to use as a key. The performance of the *MP-SA* mechanism depends on the used scheme. Using the Meghdoot scheme, the mechanism performs almost as *PP-SA*, but with a scheme with subscriptions that use the same predicate names it produces a highly skewed load distribution. This happens, because *MP-SA* chooses the same broker node for the subscriptions with the same predicate names. This shows how important for *MP-SA* is to use a subscription push method for producing good load balance. The performance of the *PP-SA* comes to the middle as it distributes the subscriptions based on the proximity of the broker with the original subscriber.

5.2.3.2 Effect of Subscribers Range

The experimental results presented in the rest of this section will focus on the *PP-SA* mechanism, as it outperforms the other two algorithms. To explore its performance with respect to the number of subscribers, we gradually increased the average number of subscribers per event and we evaluated the pub/sub system performance by delivering 100,000 events. As the number of subscribers increases, the average number of hops (as shown by Figure 5.5(a)) and the latency (as shown by Figure 5.5(b)) almost keep

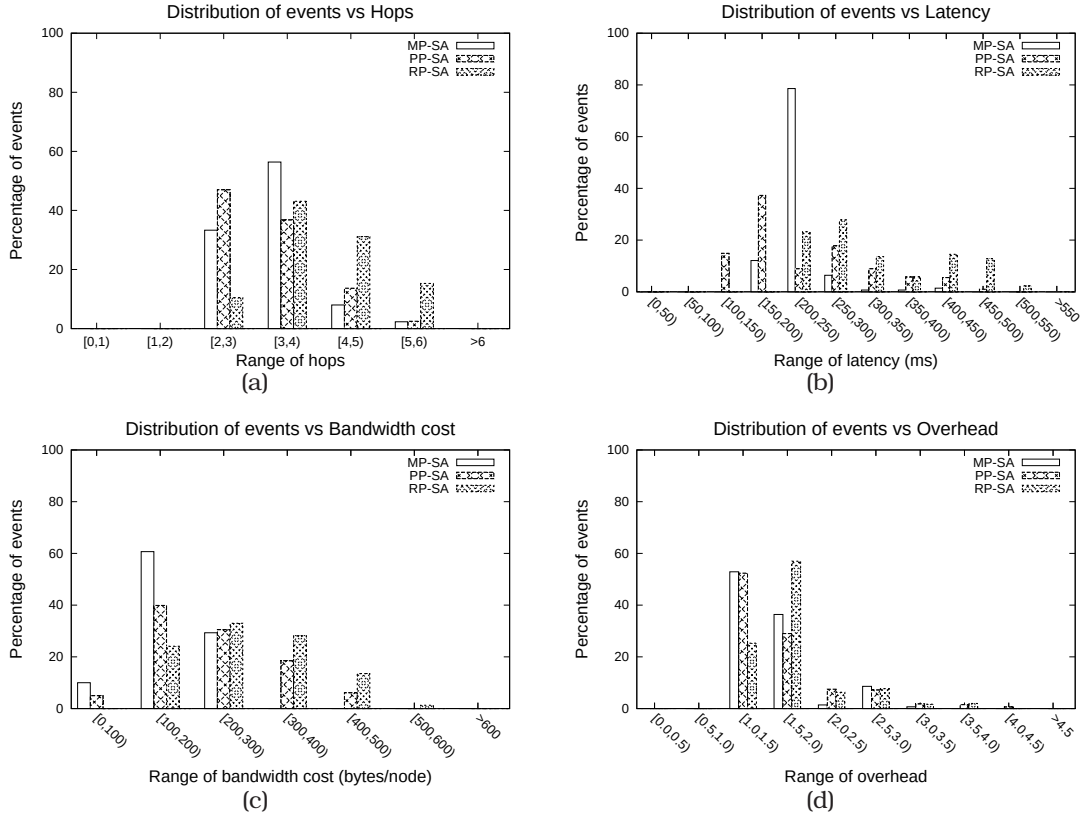


Figure 5.3: Distribution of events for *PP-SA*, *MP-SA*, and *RP-SA* mechanisms according to range of hops, latency, bandwidth, and overhead.

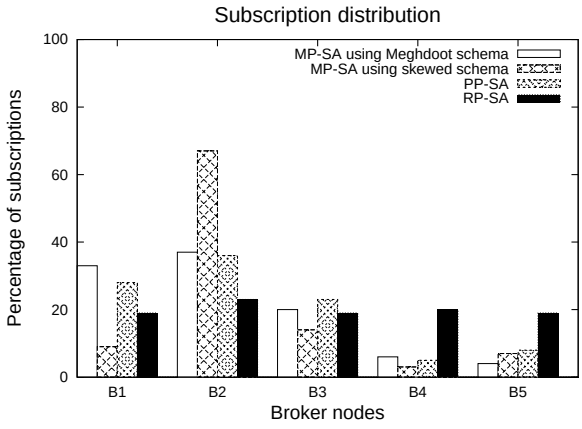


Figure 5.4: Subscription distribution in *PP-SA*, *MP-SA*, and *RP-SA* mechanisms

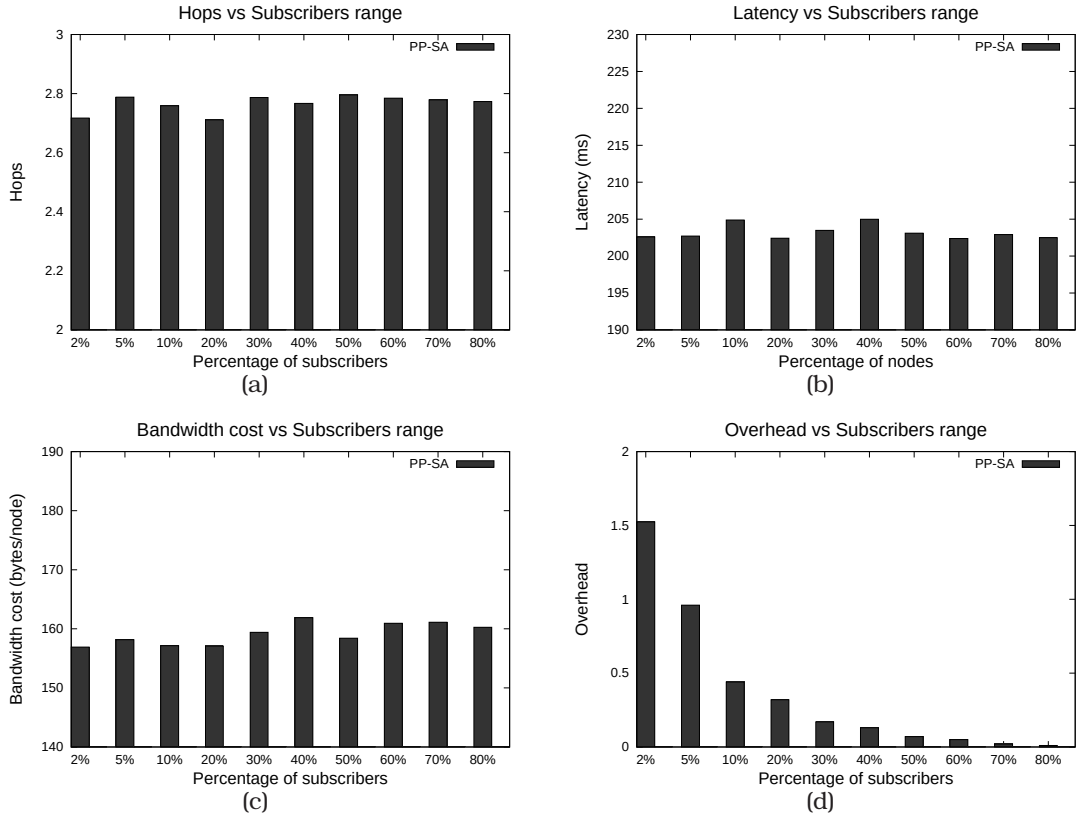


Figure 5.5: Distribution of hops, latency, bandwidth cost, and overhead vs the subscribers range for the *PP-SA* algorithm.

constant at 2.77 and 202.485 ms, respectively. On the other hand, as Figure 5.5(c) shows, the bandwidth cost increases modestly from 156.904 (bytes/node) to 160.257 (bytes/node). Nevertheless, the overhead drops significantly as shown in Figure 5.5(d). The above results indicate that the *PP-SA* algorithm can deliver events to a large number of subscribers at very low overhead, involving only a small number of intermediate nodes by the use of group message delivery. Hence, *PP-SA* proved to be very efficient in delivering events to a large number of subscribers.

5.2.3.3 Effect of Network Size

Next, we evaluated the scalability of our pub/sub mechanism based on the *PP-SA* algorithm. We present its performance under various network sizes that span between 1,024 and 65,536 nodes. For each case, we used the same configuration parameters as in the standard configuration. The required overlay hops occurred by event delivery increase modestly, as shown in Figure 5.6(a), from 2.717 hops to 4.123. This results in an increase of 66% for a 64 times network increase. The bandwidth cost (as shown in Figure 5.6(c)), and the latency (as shown in Figure 5.6(b)) increase less than double for

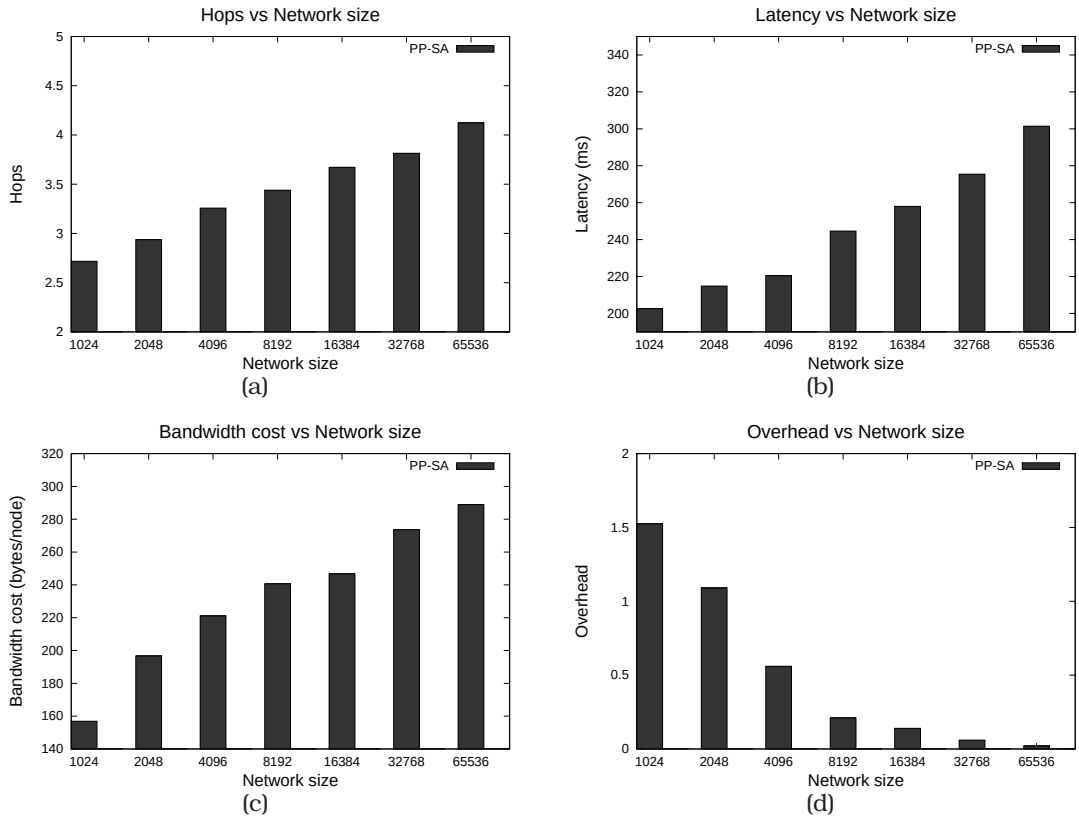


Figure 5.6: Distribution of hops, latency, bandwidth cost, and overhead vs network size for the *PP-SA* algorithm.

an 64 times increase of the network size. Furthermore, the overhead drops significantly from 1.524 to 0.02 as shown by Figure 5.6(d). The above results show that the *PP-SA* algorithm scales well with large numbers of nodes, as it requires almost the same number of intermediate nodes for the event delivery, even for high number of subscribers and network sizes.

5.3 ADORE Engine Evaluation

In this section, we present a quantitative evaluation of the ADORE distributed orchestration engine. In particular, we compared ADORE with centralized multiple-server architectures, having as main objective the assessment of the performance benefits of our architecture under high request rates. In this context, we pursued four specific goals:

1. *Performance over request rates.* The orchestration engine must effectively cope with increasing numbers of requests over a specific period of time. As the centralized

and clustered architectures become performance bottlenecks with increasing request rates, a distributed engine must be able to exploit its workers' pool and to distribute the workload among them.

2. *Performance over Web services delays*. The performance of the distributed engine must degrade gracefully for increasing Web services delays. This feature is very important when we have processes that transfer a large amount of data between Web services.

3. *Performance over network delay*. The performance of the distributed engine must not degrade substantially when the network latency increase. Therefore, we can justify its usage over a centralized engine that does not become affected by the network latency.

4. *Performance of the deployment mechanisms*. We must evaluate our deployment mechanisms namely, per-process and per-instance deployment in terms of average process execution time, throughput, and load balancing.

5.3.1 Metrics

We used four metrics to evaluate the performance and cost of our proposed architecture:

1. *Average process execution time*: $\overline{t_{reply} - t_{request}}$. The process execution time is defined as the duration from the reception of a request from the client to the transmission of the corresponding response to the client.

2. *System throughput*: $\frac{\#\text{completed instances}}{1 \text{ min}}$. The throughput is defined as the number of process instances completed per-minute.

3. *Process execution time overhead*: $\frac{T_q}{T_{scp}}$. This is defined as the ratio of the average process execution time (T_q) divided by the process execution time of a process instance in a single-server centralized system (T_{scp}).

4. *Process activities distribution load*: $\frac{\#\text{activities executed in the node}}{\#\text{total executed activities}}$. This is defined as the ratio of the process activities that are executed by a node divided by the total number of process activities executed by the system.

5.3.2 Setup

We measured the ADORE performance using the aforementioned metrics, while we varied parameters such as the request rate, the delay of the external Web services, and the network transport latency. In the standard configuration, the network consisted of 50 or 500 network nodes, with average connectivity degree k=5. The default values that we used were:

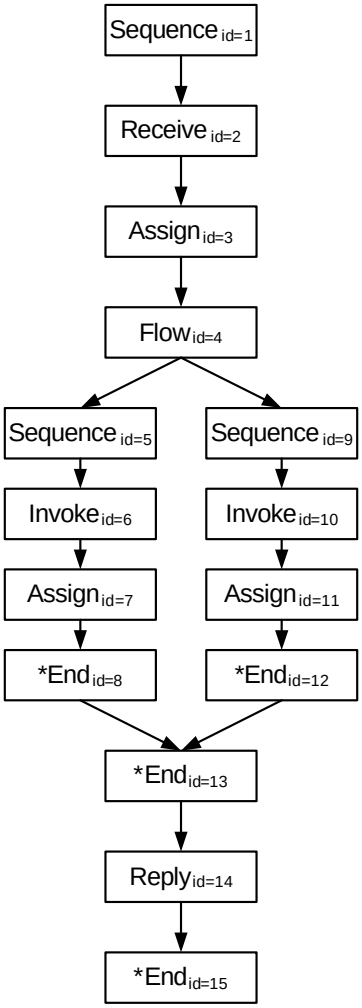


Figure 5.7: BPEL process used for the engine's evaluation.

- the request rate was 50 or 500 requests per minute
- the Web service execution time was 2,000 ms (for the invoke activity)
- the activities execution time was 50 ms
- the transmission latency was defined between 100 ms and 500 ms.

We performed all experiments using the BPEL process[9] presented in Figure 5.7.

[9]In our implementation we used special <end> activities that are not part of the official BPEL specification. These activities can be omitted using a different pub/sub translation model. We deployed the <end> activities to the same nodes that we used for the proceeding activities. This way, we reduced the latency overhead that was introduced by our translation model.

In our simulation, new nodes were joining into the system until the total number of nodes was reached (i.e. 50 or 500 nodes). After the system stabilization, we deployed the BPEL process to a number of nodes using per-instance deployment and we simulated requests from external clients to create process instances. The latter, are modeled using the Poison distribution. We evaluated our system by comparing its efficiency with four different centralized multiple-server systems:

1. *M/M/1-50*. A centralized multiple-server system with exponential service time distribution (M/M/1). This system consists of 50 servers that use different arrival queues. This system can handle a maximum request rate of 700 events per-minute.

2. *M/M/1-145*. A centralized multiple-server system with exponential service time distribution (M/M/1). This system consists of 145 servers that use different arrival queues. This system can handle a maximum request rate of 2,000 events per-minute.

3. *M/D/1-50*. A centralized multiple-server system with constant service time distribution (M/D/1). This system consists of 50 servers that use different arrival queues. This system can handle a maximum request rate of 700 events per-minute.

4. *M/D/1-145*. A centralized multiple-server system with constant service time distribution (M/D/1). This system consists of 145 servers that use different arrival queues. This system can handle a maximum request rate of 2,000 events per-minute.

Additionally, we assumed that the following statements always hold for the above systems: a) the requests follow a Poison arrival rate, b) the dispatching principle does not give any preference to items based on service times, c) the formulas for standard deviation assume first-in, first-out dispatching (FIFO), and d) no items are discarded from the queue.

Furthermore, in multiple-server systems with λ arrival rate, N number of nodes, and T_s mean service time for each arrival, their utilization is defined as:

$$\varrho = \frac{\lambda * T_s}{N} \tag{5.1}$$

For a multiple-server M/M/1 system the average process execution time (T_q) is:

$$T_q = \frac{T_s}{1 - \varrho} \tag{5.2}$$

For a multiple-server M/D/1 system the average process execution time (T_q) is:

$$T_q = \frac{T_s(2 - \varrho)}{2(1 - \varrho)} \tag{5.3}$$

The theoretical maximum input rate (λ_{max}) for M/M/1 and M/D/1 multiple-server systems with N nodes is:

$$\lambda_{max} = \frac{N}{T_s} \tag{5.4}$$

5.3.3 Experimental Results

In the next sections, our evaluation tries to find the best and worst conditions for our engine's operation using varied simulation parameters, in an attempt to identify the cases for which our architecture is well suited.

5.3.3.1 Performance with varied Request Rate

This experiment varied the process invocation rate, where each invocation generated a process instance. We measured the average process execution time, the throughput, and the process execution time overhead.

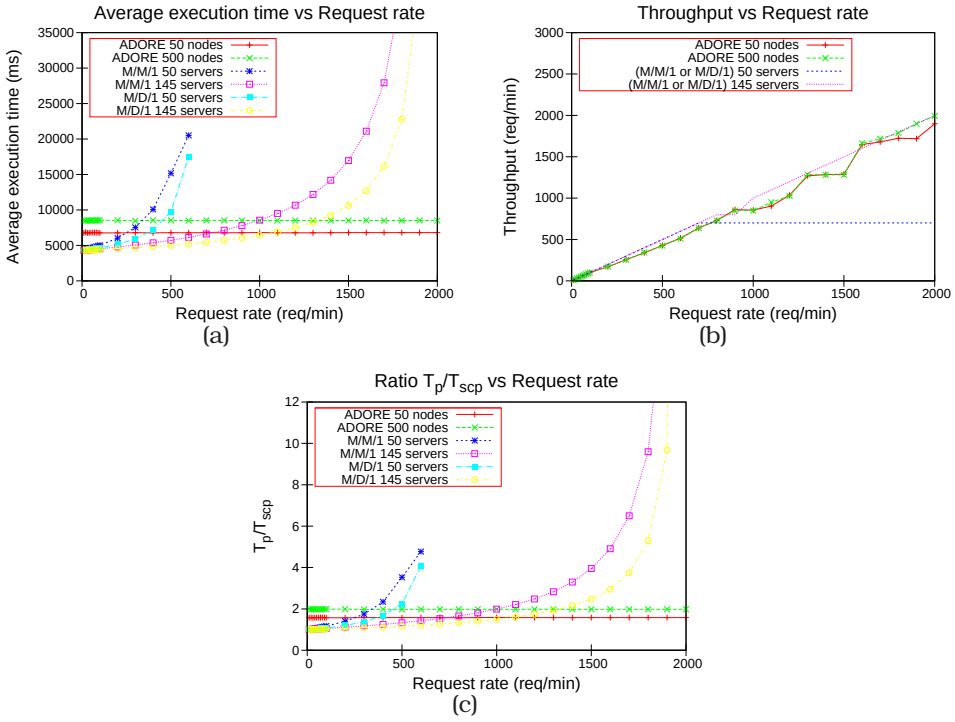


Figure 5.8: ADORE performance with increasing request rate.

As shown in Figure 5.8(a), for lower requests rates, the multiple-server approach offers better average process execution times (the ADORE requires 1.5 times more average execution time). This is caused by the communication overhead of traversing the DHT network using pub/sub messages for control and data flow. In the distributed setup these costs are not negligible and have an important impact for lower request rates.

As the request rate increases, the ADORE engine provides more or less the same performance while for the multiple-server approaches the required average execution time increases exponentially. Nevertheless, when the request rates are higher than 1000 requests per-minute, the distributed architecture outperformed the centralized ones. This happens, because the centralized architectures starts to create larger queues for handling the requests while the ADORE deployment mechanisms takes advantage of the multiple available nodes and distributes the activities to them.

The throughput is presented in Figure 5.8(b). The distributed architecture exhibits almost optimal throughput. It outperforms the multiple-server engines with 50 nodes and as the request rate increases it has the same performance with the 500 nodes multiple-server approach (that was designed for handling requests rates of 2000 requests per-minute). Note that, for request rates lower that 700 requests per-minute, the throughput of all approaches is almost the same as the request rate because none of the approaches has reached its maximum throughput.

Figure 5.8(c) presents the ratio of the average process execution time in all configurations via a process instance execution time in a single centralized engine ($T_{scp} = 4300$ ms). This diagram shows that our engine requires twice the time accomplished by the singe-server instance case, without being affected by the request rate. This proves that the throughput performance that is gained does not inflict a significant increase of the average process execution time. This overhead is constant and is unaffected by the request rate increase.

5.3.3.2 Performance with varied Web Service Delay

To better understand the effect of invoking external Web services on both the process execution time and the throughput, we performed a number of experiments using three different requests rates (50 req/min, 500 req/min, and 1000 req/min), while varying the Web service delay from 200 ms to 8000 ms.

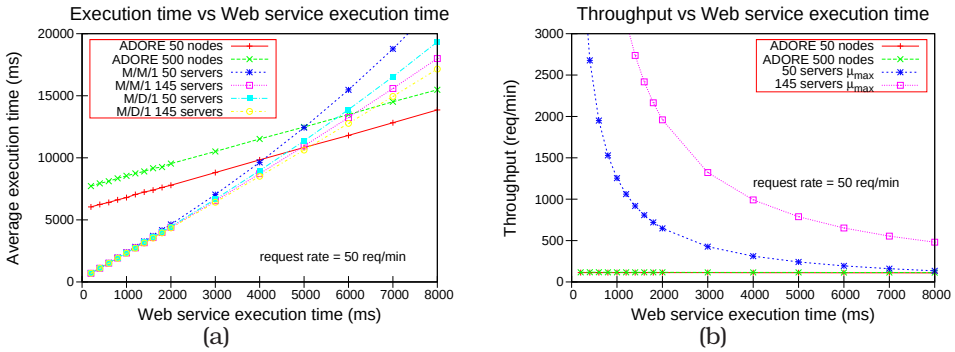


Figure 5.9: ADORE performance vs Web service delay (50 req/min).

With low request rates (50 req/min), the results in Figure 5.9(a) show that a longer Web service delay increases the average execution time for all three deployment scenar-

ios. When the delay is small the multiple-server approaches perform best, by avoiding the communication overhead that is present in ADORE. On the other hand, when the Web service delays increase, the distributed approach performs best, requiring 24% less time per-process instance. The throughput in low requests rates increases slightly for the ADORE, while for the multiple-server approaches the theoretical maximum throughput drops significantly and converges with the ADORE for larger Web service delays.

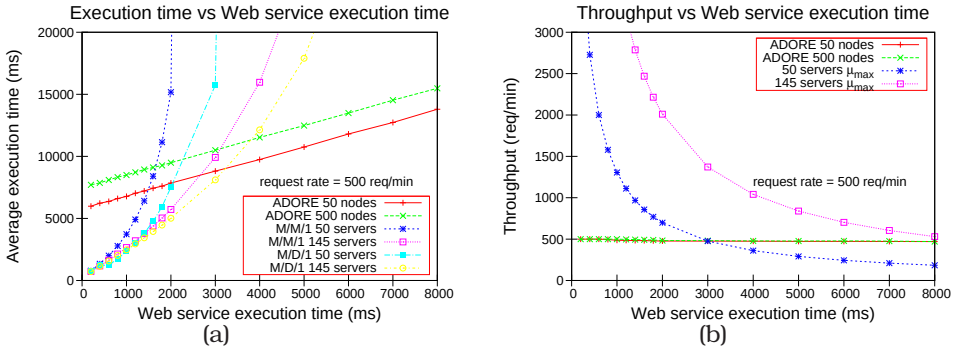


Figure 5.10: ADORE performance vs Web service delay (500 req/min).

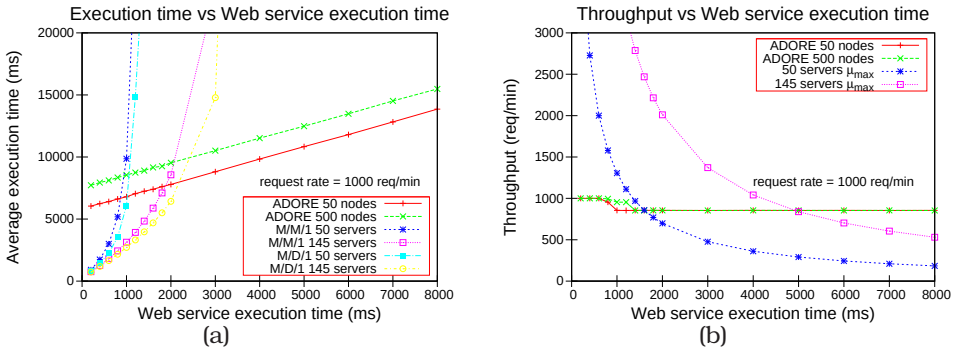


Figure 5.11: ADORE performance vs Web service delay (1000 req/min).

With medium request rates (500 req/min), the results in Figure 5.9(a) show that, a longer Web service delay increases the average execution time for all three deployment scenarios. Nevertheless, this time ADORE scales better than the multiple-server architectures. Note that, a 40 times increase in the Web service delay corresponds to a linear increase for ADORE (it requires only 2.3 times more average process execution time). On the contrary, it corresponds to an exponential increase for the multiple-server approaches. As shown in Figure 5.11(a), the same trend continues for even larger request rates (1000 req/min), where ADORE outperforms the multiple-server approaches. The throughput in both cases (Figure 5.10(b) and Figure 5.11(b)) remains constant for ADORE as the Web service delay increases, while for the multiple-server approaches the maximum theoretical throughput drops quickly and converges with that of ADORE.

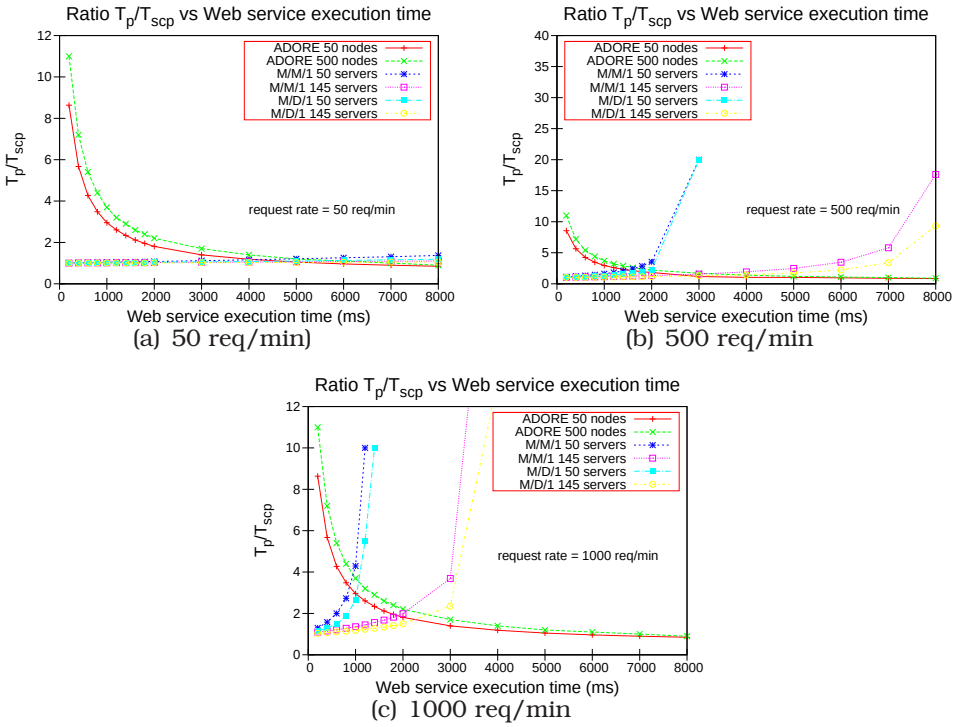


Figure 5.12: ADORE performance average execution time vs single server execution time with increasing Web service delay.

Finally, Figure 5.12 presents the process execution time overhead in all cases. We see that ADORE has greater overhead due to network delays. Nevertheless, as the Web service delay increases, it provides the best performance, since it requires almost the same average process execution time with the single centralized approach. On the other hand, the performance of the centralized multiple-server approach degrades significantly, as the average process execution time increases exponentially with the Web service delay.

5.3.3.3 Performance with varied Latency

It is crucial for the performance of the ADORE engine to degrade gracefully as the network latency increases. For this reason, we performed a set of experiments and measured the average process execution time and throughput while we were increasing the network latency from 100 ms to 1000 ms. This overhead was applied to any transmission that took place in the DHT network and thus it affected all messages. We made measurements using three different request rates (100 req/min, 500 req/min, and 1000 req/min).

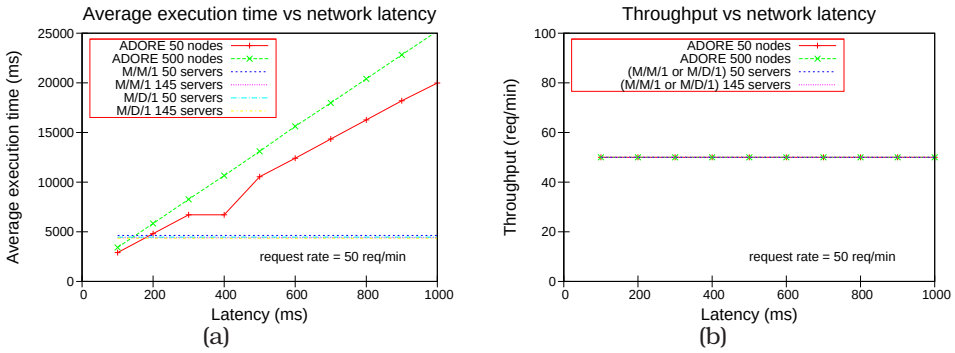


Figure 5.13: ADORE performance vs network latency (50 req/min).

Figure 5.13 presents the performance of ADORE for low request rates and increasing latency. For latency lower than 200 ms, ADORE achieves the same average process execution time as the multiple-server approaches. As the network latency increases the average process execution time of ADORE increases linearly (Figure 5.13(a)). Furthermore, Figure 5.13(b) shows that at 50 req/min, the throughput of ADORE performance remains unaffected by the increasing latencies and equals to the optimal.

For medium request rates (Figure 5.14), the average process execution time of ADORE increases linearly with increasing latency. As shown by Figure 5.14(a), for network latency lower than 600 ms ADORE exhibits better average process execution time than the multiple-server approaches. Furthermore, Figure 5.13(b) shows that at 500 req/min, the throughput of ADORE performance degrades by 44% while the network latency increases tenfold.

Finally, for high request rates (see Figure 5.15) the average process execution time of ADORE again increases linearly with increasing latency. As shown in Figure 5.15(a),

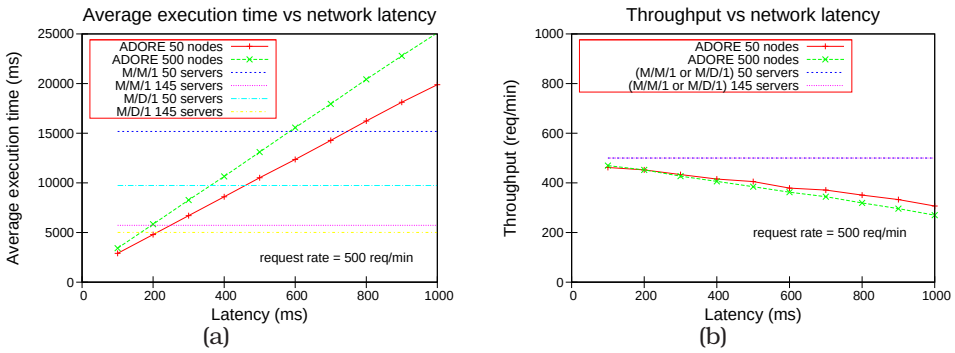


Figure 5.14: ADORE performance vs network latency (500 req/min).

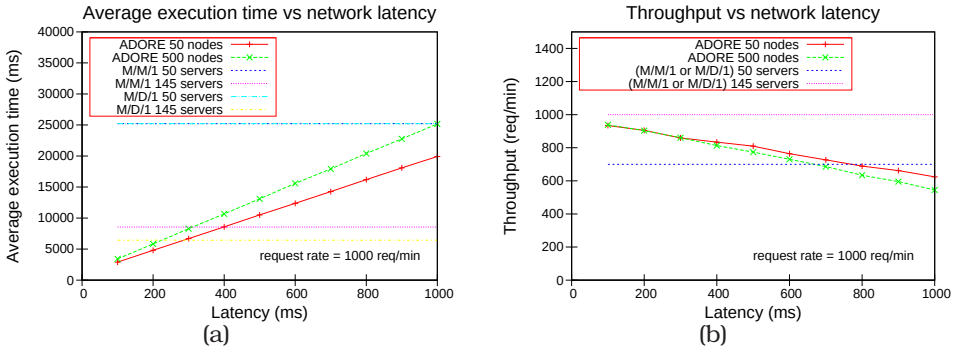


Figure 5.15: ADORE performance vs network latency (1000 req/min).

ADORE outperforms the 50 multiple-server approaches but looses by the 140 multiple-server approach. As far as the throughput is concerned, at these rates the throughput of ADORE performance again degrades by 44% while the network latency increases ten times (as shown in Figure 5.15(b)).

5.3.3.4 Per-process vs Per-instance Deployment

We compared the performance of the two deployment mechanisms: *per-process* and *per-instance* deployment, in terms of average process execution time and node load. Figure 5.16(a), shows a comparison of the average process execution time required by the two mechanisms over increasing request rates. The two mechanisms exhibit the same performance, while the per-instance deployment requires on average 8% more time. This behavior continued as we studied their behavior by increasing the network latency. As shown in Figure 5.16(b), the per-instance deployment mechanism, produces slightly bigger overhead and only for high network latency.

On the other hand, the per-instance deployment has better load balancing behavior. In extreme cases, as Figure 5.17 shows, as we created instances of the same BPEL process the per-process deployment used only a small fraction of the worker nodes to execute

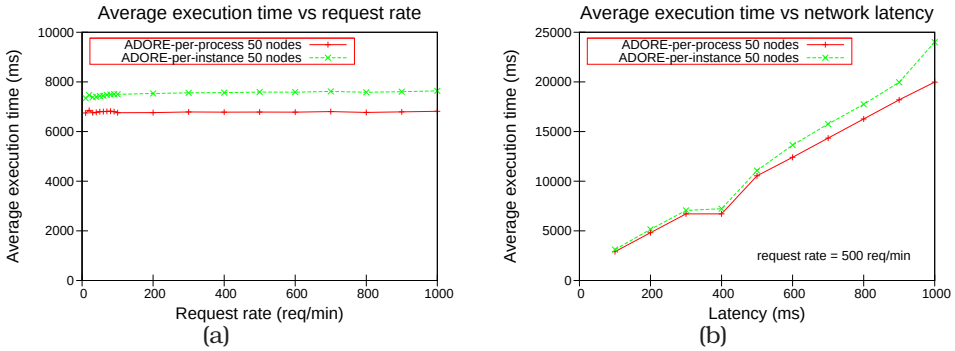


Figure 5.16: ADORE performance using per-process vs per-instance deployment with increasing request rate.

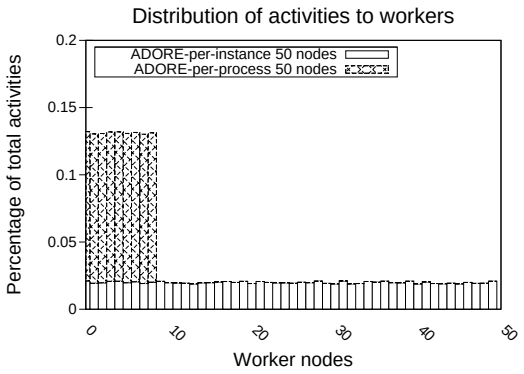


Figure 5.17: Activity distribution in per-process and per-instance deployment.

activities. As opposed to that, the per-instance deployment used all the workers in a round-robin fashion. We believe that it is worth paying a small cost in order to have better load balancing in the DHT network and avoid hot spots that could degrade the engine's performance. Our results seem to suggest that, the per-instance deployment is overall a better solution. Nevertheless, we need to perform more experiments by studying the engine's performance in real network conditions and under real use cases scenarios. This work is left for the future.

5.4 Conclusions

In this chapter, we provided a throughout evaluation of our pub/sub mechanism and ADORE's performance. Our experiments have led to the following conclusions:

1. Our proposed pub/sub mechanisms scale well and their effectiveness does not degrade over increasing number of subscribers and network sizes. Furthermore, our mechanisms produce low network overhead and require minimal number of hops for event dissemination. Moreover, our algorithms yield good load balancing over the broker nodes. We conclude that the best deployed algorithm is the *PP-SA* algorithm.

2. Our orchestration engine can effectively cope with increasing request rates. Moreover, for network latencies between 150 and 500 ms, the engine provides optimal throughput performance with very good average process execution times. In all the above cases, it outperforms the centralized multiple-server approaches in terms of average process execution time, throughput, and overhead. Its performance decreases gracefully with increasing Web services delays and network latencies. Finally, the per-instance deployment mechanism provides better load balancing features with only 8% overhead in the average process execution time comparatively with the per-process deployment.

Chapter 6

Conclusions and Future Work

This chapter gives a summary of our proposal and presents a number of future directions for our work.

6.1 Conclusions

In this thesis, we proposed a distributed orchestration engine named ADORE, that is capable of deploying and executing BPEL processes. We presented in detail the engine's design, architecture, and described the operation of its key components.

ADORE, is based on a pub/sub infrastructure and uses a number of loosely coupled light-weight nodes, arranged over a DHT network, to carry out the business process execution. In short, ADORE translates the BPEL processes descriptions into a set of pub/sub messages that are stored, matched, and published over the DHT nodes. The pub/sub layer is based on the content-based pub/sub model and provides simplified and efficient interaction among the nodes by exploiting network proximity metrics. We analyzed the proposed pub/sub layer algorithms for subscription and publication storage, matching, and propagation. Our algorithms are designed to provide good load balance and produce minimum overhead.

The engine's DHT nodes belong into three categories: a) deployer nodes that map BPEL process into pub/sub messages, b) workers that execute BPEL activities by using pub/-sub messages, and c) brokers that provide pub/sub matching and route the messages over the DHT network. Using the aforementioned nodes and the characteristics of the pub/-sub layer our engine is capable of executing efficiently multiple process instances. We described the engine's operation during its five key operations: a) startup, b) deployment, c) execution, d) redeployment, and e) undeployment. During each phase we described the pub/sub messages that are sent and we gave full details of the nodes interactions and operation.

Finally, we made an evaluation of the proposed pub/sub mechanisms and the engine's operation as a whole by comparing our distributed engine with a number of centralized multiple-server engines. The evaluation indicates that there are performance benefits

from our distributed approach that are more apparent under high process request workloads. In short:

- Our pub/sub mechanisms produce low network overhead and require minimal number of hops for event dissemination. Moreover, our algorithms produce good load balancing over the broker nodes.
- Our orchestration engine can effectively cope with increasing request rates (over 500 req/min). Furthermore, for network latencies between 150 and 500 ms our engine provides optimal throughput performance with very good average process execution times. In all the above cases, it outperforms the centralized multiple-server approaches in terms of average process execution time, throughput, and overhead. Moreover, its performance decreases gracefully with increasing Web services delays and network latencies.

6.2 Future Work

First, we would like to produce more experiments with larger business processes and broker topologies, using a real implementation. Furthermore, we want to evaluate our engine contrast a similar approach, such as NIÑOS [LMJ10].

Second, we want to study the performance consequences of a different deployment algorithm that will try to deploy the activities to worker nodes that have better proximity (e.g. have network connections with lower latency) with the external clients and Web services. This way, we could benefit from the different characteristics of our DHT nodes and reduce the latency costs and the average process execution time. In this direction, there are a number of approaches that we could benefit from and use them in our implementation [SPvS04, SPPvS08, GSG02, VKK07]. As far as we know, none of the current orchestration engines follows such an approach or tries to do something similar.

Finally, we look forward to apply our solution to domains that require long running business processes that manipulate large amount of data. This scenario is very common in the environmental services and there is outgoing research in this area[1]. Moreover, in this case the distributed orchestration engine must be able to cope with node failures. In this case it must provide a mechanism that will use methods for data replication in order to restore the state of the process in execution, when an failure happens.

[1] http://www.envision-project.eu/

Appendix A

Mapping BPEL to the Publish/Subscribe Language

A.1 Mapping Basic Activities

A.1.1 <receive> activity

The <receive> activity provides services to the partners of the BPEL business process. If the <receive> activity has a createInstance element set to yes, then when a request is received from a partner, the BPEL engine creates a new business process instance[1]. Once a process instance is created, the business process can continue its execution and perform other basic or structured activities. In this case any other activity in the same business process instance loses its ability to create new business process instances.

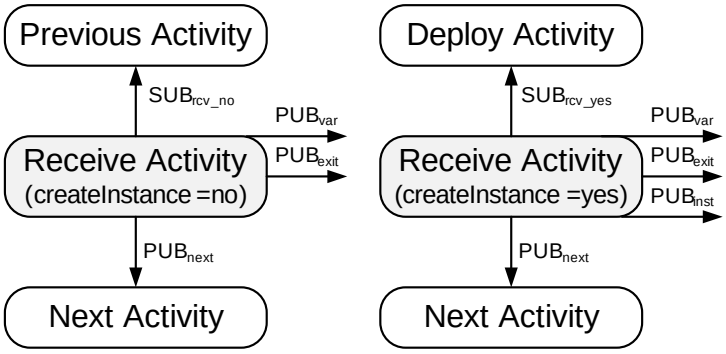
Listing A.1: BPEL syntax for the <receive> activity.

```
<receive partnerLink="NCName"
         portType="QName"?
         operation="NCName"
         variable="BPELVariableName"
         createInstance="yes|no"?
         standard-attributes/>
    standard-elements
</receive>
```

Listing A.2: <receive> example; a client can invoke the Print operation provided by the BPEL process via the PrintServiceInterface. The PrintRequest variable stores the received data.

```
<receive partnerLink="Client">
         portType="PrintServiceInterface"
         operation="Print"
```

[1]Only <receive> and <pick> activities (see section A.2.4) exhibit this behavior.



(a) Receive activity createInstance=NO.

(b) Receive activity createInstance=YES.

Figure A.1: Publish and subscribe messages for the receive activity. The Figure A.1(a) presents the messages when the <createInstance=No> while Figure A.1(b) presents the messages in a receive activity with <createInstane=yes>

```
        variable="PrintRequest" />
```

<receive> activity subscription messages

$SUB_{rcv_{yes}}$= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*activityID*, eq, "activityId"], [*variableName*, eq, "BPELVariableName"],
[*partnerLink*, eq, "NCName"], [*portType*, eq, "QName"]?,
[*operation*,eq, "NCName"]

$SUB_{rcv_{no}}$= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*partnerLink*, eq, "NCName"], [*portType*, eq, "QName"]?,
[*operation*, eq, "NCName"]

<receive> activity publication messages

$PUB_{instance}$= [*class*, INSTANCE], [*process*, "processName"], <<instanceId>>

PUB_{var}= [*class*, VARIABLE_UPDATE], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "activityId"],
[*variableName*, "BPELVariableName"], <<variableData>>

PUB_{next}= publication to the next activity

<receive> activity subscription and publication algorithm

If the <receive> activity has the create instance element set to *yes* then the worker agent must subscribe to $SUB_{rcv_{yes}}$. Otherwise the agent must subscribe to $SUB_{rcv_{no}}$.

Algorithm 21: <receive> activity algorithm

```
if createInstance == yes then
    subscribe to SUB_{rcv_yes}
    publish {PUB_instance, PUB_var, PUB_next}
else
    subscribe to SUB_{rcv_no}
    publish {PUB_var, PUB_next}
end if
```

After it is triggered with the appropriate publication the worker agent executes the receive operation. If the receive is successfully executed, the worker publishes PUB_{var}, with the updated variable information, and PUB_{next} that triggers the next activity. If the create instance attribute was *yes* the agent also publishes $\text{PUB}_{instance}$ that informs the deployer and all the process activity workers about the new process instance id. Finally, in case the operation fails, a standard error publication is published.

A.1.2 <reply> activity

The <reply> activity sends a response from the BPEL process to a client. This response is send as a reply to a previous request that was accepted through a <receive> activity. Thus the <reply> activity is used for implementing synchronous request-response interactions. When the BPEL process requires asynchronous request-response interactions, these are facilitated using <invoke> activities (see section A.1.3).

Listing A.3: BPEL syntax for the <reply> activity.

```
<reply partnerLink="NCName"
       portType="QName"?
       operation="NCName"
       variable="BPELVariableName"
       standard-attributes>
    standard-elements
</reply>
```

Listing A.4: <reply> example; the BPEL process calls the client's TravelApproval operation using as input the TravelResponse variable.

```
<reply partnerLink="client"
       portType="TravelApprovalPT"
       operation="TravelApproval"
       variable="TravelResponse" />
```

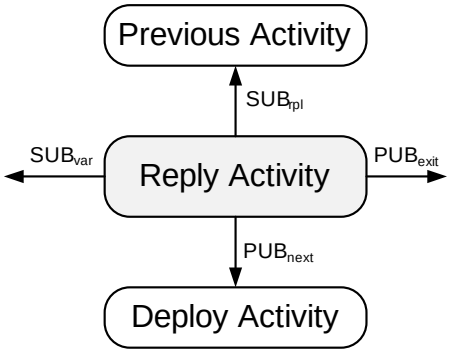


Figure A.2: Publish/Subscribe messages for the <reply> activity.

<reply> activity subscription messages

SUB_{reply}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*partnerLink*, eq, "NCName"], [*portType*, eq, "QName"]?,
[*operation*, eq, "NCName"]

SUB_{var}= [*class*, eq, VARIABLE-UPDATE], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*variableName*, eq, "BPELVariableName"]

<reply> activity publication messages

PUB_{reply}= [*class*, REPLY], [*process*, "processName"],
<<instanceId, BPELVariableName, variableData>>

<reply> activity subscription and publication algorithm

The agent responsible for the <reply> activity must subscribe to $SUB_{reply} \wedge SUB_{var}$. If the operation is successfully executed, the worker publishes PUB_{reply} that carries the response data. This publication targets the deployer node that handles the interaction with the client. If the operation fails, a standard error publication is published.

A.1.3 <invoke> activity

The <invoke> activity is used by the BPEL process to invoke (synchronous or asynchronous) its partners' web service operations. Furthermore, it is used to implement the asynchronous request-response pattern, as the BPEL process can use <invoke> to return the results to its caller.

Listing A.5: BPEL syntax for the <invoke> activity.

```
<invoke partnerLink="NCName"
        portType="QName"?
        operation="NCName"
        inputVariable="BPELVariableName"?
        outputVariable="BPELVariableName"?
        standard-attributes>
    standard-elements
</invoke>
```

Listing A.6: An example using an asynchronous <invoke>; the BPEL process request information about a flight availability from a web service.

```
<invoke name="FlightAvailabilityAyncInv"
        partnerLink="FlightAvailabilityPL"
        portType="FlightAvailabilityPT"
        operation="FlightAvailability"
        inputVariable="FlightDetails"
/>
```

Listing A.7: An example using a synchronous <invoke>; the BPEL process makes a synchronous request-response call about the flight status.

```
<invoke name="FlightStatusSyncInv"
        partnerLink="FlightStatusPL"
        portType="FlightStatusPT"
        operation="FlightStatus"
        inputVariable="FlightStatusRequest"
        outputVariable="FlightStatusResponse"
/>
```

<invoke> activity subscription messages

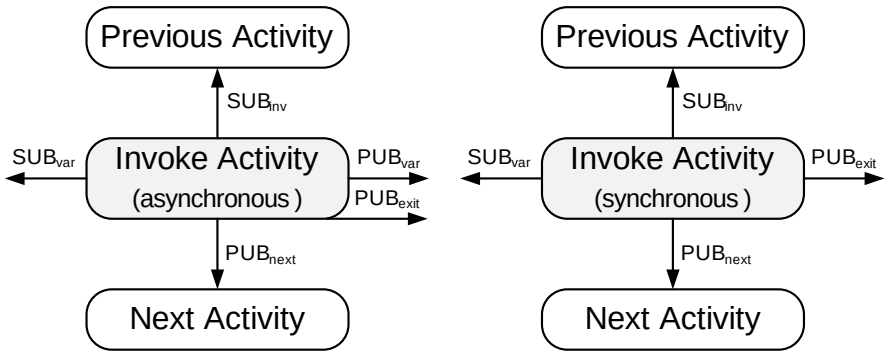
SUB_{inv}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*portType*, eq, "QName"]?, [*operation*, eq, "NCName"]

SUB_{var}= [*class*, eq, VARIABLE-UPDATE], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*variableName*, eq, "BPELInputVariableName"]

<invoke> activity publication messages

PUB_{var}= [*class*, VARIABLE-UPDATE], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "activityId"],
[*variableName*, "BPELOutputVariableName"], <<variableData>>

PUB_{next}= publication to the next activity



(a) Invoke activity asynchronous. (b) Invoke activity asynchronous.

Figure A.3: Publish/Subscribe messages for the invoke activity. Figure A.3(a) presents an asynchronous invocation while Figure A.3(b) shows the messages involved in a synchronous invocation.

<invoke> activity subscription and publication algorithm

Algorithm 22: <invoke> activity algorithm

if $\exists$ inputVariable **then**
  subscribe to $\text{SUB}_{inv} \wedge \text{SUB}_{var}$
else
  subscribe to SUB_{inv}
end if
if $\exists$ outputVariable **then**
  wait for successful execution
  publish $\{\text{PUB}_{var}, \text{PUB}_{next}\}$
else
  publish PUB_{next}
end if

If the <invoke> activity requires an input variable, then the worker agent must subscribe to $\text{SUB}_{inv} \wedge \text{SUB}_{var}$. Otherwise the worker agent subscribes only to SUB_{inv}. After the worker is triggered by the proper publications, it invokes the associated web service operation. If the invocation is synchronous the agents waits for the operation to execute successfully and then publishes PUB_{var} and PUB_{next}. If the invocation is asynchronous, the agent only to publishes PUB_{next} to trigger the following activity. If the operation fails, the worker publishes a standard error publication.

A.1.4 <assign> activity

The <assign> activity is used: a) to copy data from one variable to another and b) to construct and insert new data using expressions and literal values.

Listing A.8: BPEL syntax for the <assign> activity.

```
<assign standard-attributes>
        standard-elements
        <copy>+
            from-spec
            to-spec
        </copy>
</assign>

<!--examples of from-spec -->
<from variable="name" part="cname"?/>
<from variable="name" part="cname"? query="queryString"?/>
<from variable="name" property="qname"/>
<from expression="general-expr"/>
<from> literal value </from>

<!-- examples of to-spec -->
<to variable="name" part="cname">
<to variable="name" part="cname"? query="queryString"?/>
<to variable="cname" property="qname"/>
```

Listing A.9: An <assign> example were the BPEL process assigns the surname from an undergraduate student to a postgraduate student.

```
<assign name="myAssign">
 <copy>
  <from variable="undergradStudent" part="surname" />
  <to variable="postgradStudent" part="surname" />
 </copy>
</assign>
```

<assign> activity subscription messages

SUB_{asgn}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

SUB_{var}= [*class*, eq, VARIABLE-UPDATE], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*variableName*, eq, "BPELVariableName"]

<assign> activity publication messages

PUB_{var}= [*class*, VARIABLE-UPDATE], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "activityId"],
[*variableName*, "BPELVariableName"], <<variableData>>

PUB_{next}= publication to the next activity

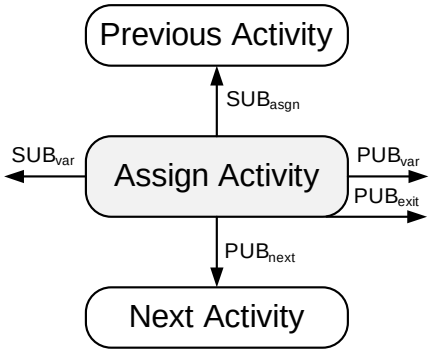


Figure A.4: Publish/Subscribe messages for the <assign> activity.

<assign> activity subscription and publication algorithm

If the from-spec is a constant expression like a: string, integer, boolean, literal, or partnerLink attribute, the constant expression will be directly injected to the <assign> activity. This way the worker needs to subscribe only to SUB_{asgn}. Otherwise, if the from-spec is a variable or expression involving a variable part or variable property, the worker agent responsible for the <assign> has to subscribe to the content of from-spec. Thus the worker does a composite subscription $\text{SUB}_{asgn} \wedge \text{SUB}_{var}$. When the operation is successfully executed, the worker publishes the PUB_{var} that carries the updated variable value and triggers the next activity via PUB_{next}. If the operation fails, then a standard error publication is published.

A.1.5 <exit> activity

The <exit> activity terminates a business process instance.

Listing A.10: BPEL syntax for the <exit> activity.

```
<exit standard-attributes>
      standard-elements
</exit>
```

Listing A.11: <exit> example; the process terminates its execution.

```
<exit/>
```

<exit> activity subscription messages

SUB_{exit}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

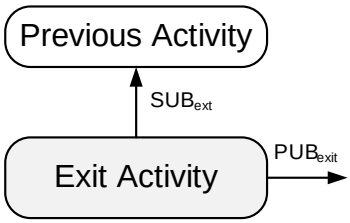


Figure A.5: Publish/Subscribe messages for the <exit> activity.

<exit> activity publication messages

PUB$_{exit}$= [*class*, EXIT], [*process*, "processName"],
<<instanceID, "SUCCESS|FAILURE">>

<exit> activity subscription and publication algorithm

The <exit> worker agent subscribes to SUB$_{exit}$. If the worker is triggered by the appropriate publication message it publishes PUB$_{exit}$, carrying the process instance id and a "SUCCESS" message. If the operation fails, then a standard error publication is published.

A.1.6 <empty> activity

The <empty> activity does nothing.

Listing A.12: BPEL syntax for the <exit> activity.

```
<empty standard-attributes>
       standard-elements
</empty>
```

Listing A.13: <empty> activity example; the process does nothing.

```
<empty/>
```

<empty> activity subscription messages

SUB$_{emp}$= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

<empty> activity publication messages

PUB$_{next}$= publication to the next activity

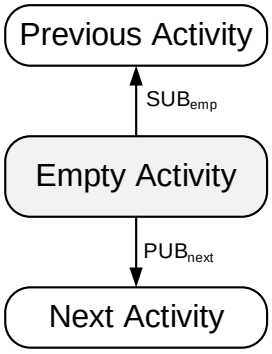


Figure A.6: Publish/Subscribe messages for the <empty> activity.

<empty> activity subscription and publication algorithm

The <empty> worker agent subscribes to SUB_{emp}. If the agent is triggered by the proper publication, it publishes PUB_{next}. If the operation fails, then a standard error publication is published.

A.1.7 <end> activity

The <end> activity is a special version of the empty activity and is used by our protocol in some structured activities. It is used to transfer control to the first worker of a structured activity.

<end> activity subscription messages

SUB_{prev_i}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

<end> activity publication messages

PUB_{next}= publication to the next activity

<end> activity subscription and publication algorithm

The <end> worker agent subscribes to the publications of the proper previous activities using SUB_{prev_i}. If the agent is triggered by all the associated parent publications, it publishes PUB_{next}. If the operation fails, then a standard error publication is published.

A.1.8 <wait> activity

The <wait> activity is used to delay the BPEL process execuction for a certain period of time, or to stall the process until a certain deadline is reached.

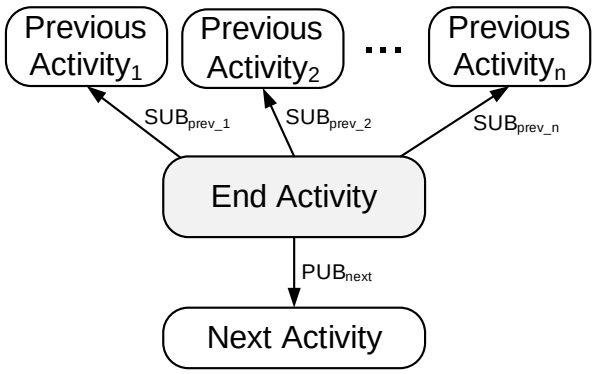


Figure A.7: Publish/Subscribe messages for the <end> activity.

Algorithm 23: <end> activity algorithm

for all previous activities $prev_i \in [1, n]$ **do**
  subscribe to SUB_{prev_i}
end for
if [$\text{SUB}_{prev_1} \wedge \text{SUB}_{prev_2} \wedge \ldots \wedge \text{SUB}_{prev_n}$] are triggered **then**
  publish PUB_{next}
end if

Listing A.14: BPEL syntax for the <wait> activity.

```
<wait (for="duration-expr" | until="deadline-expr") standard-attributes>
    standard-elements
<wait/>
```

Listing A.15: In the first example the <wait> activity stalls the execution until eight o'clock in GMT+2 timezone; in the second example the <wait> activity stalls the execution for thirty seconds.

```
<wait until="2011-06-1T8:00:00+2:00"/>
<wait for="P30S">
```

<wait> activity subscription messages

SUB_{wait}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

<wait> activity publication messages

PUB_{next}= publication to the next activity

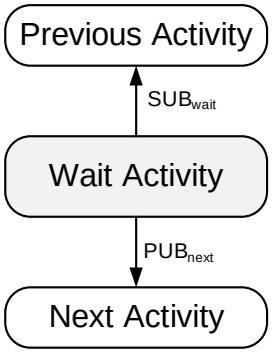


Figure A.8: Publish/Subscribe messages for the <wait> activity.

<wait> activity subscription and publication algorithm

The worker agent subscribes to SUB$_{wait}$. If the agent receives the proper publication message, it stalls the execution for a specific time interval or until a certain deadline. Afterwards, the worker publishes PUB$_{next}$. If the operation fails, then a standard error publication is published.

A.2 Mapping Structured Activities

A.2.1 <sequence> activity

The <sequence> activity is used to define a group of activities that will be executed in a sequential order.

Listing A.16: BPEL syntax for the <sequence> activity.

```
<sequence standard-attributes>
    standard-elements
    activity+
</sequence>
```

Listing A.17: <sequence> example; first is executed the <receive> activity then the <assing> activity and last the <invoke> activity.

```
<sequence standard-attributes>
    <receive .../>
    <assign .../>
    <invoke .../>
</sequence>
```

<sequence activity subscription messages

SUB_{seq}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

SUB_{fin}= [*class*, eq, END-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "EndActivityId"]

<sequence> activity publication messages

PUB_{inner_1}= publication to first inner activity

PUB_{next}= publication to next activity

<end> activity subscription messages

SUB_{end}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "LastInnerActivityId"]

<end> activity publication messages

PUB_{fin}= [*class*, END-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "EndActivityId"],

<sequence> activity subscription and publication algorithm

Algorithm 24: <sequence> activity algorithm

```
subscribe to SUB_seq ∨ SUB_fin
if SUB_seq was triggered then
   publish PUB_inner1
else if SUB_fin was triggered then
   publish PUB_next
end if
```

The worker responsible for the <sequence> activity must subscribe to the activity that preceded the <sequence> and to the activity <end>. The latter represents the end of the <sequence> body execution. Each activity in the body of the <sequence> subscribes to the previous inner activity and publishes to the next inner activity, with two exceptions: a) the first inner activity subscribes to the <sequence> activity, and b) the last inner activity publishes to the <end> activity.

Thus the <sequence> worker subscribes to SUB_{seq} and waits for a proper trigger publication. After it is triggered, the worker sends a publication to the first inner activity in the body of the <sequence>. We must also notice that the <sequence> agent also subscribes to the <end> activity (SUB_{fin}). This way it can be triggered after the last

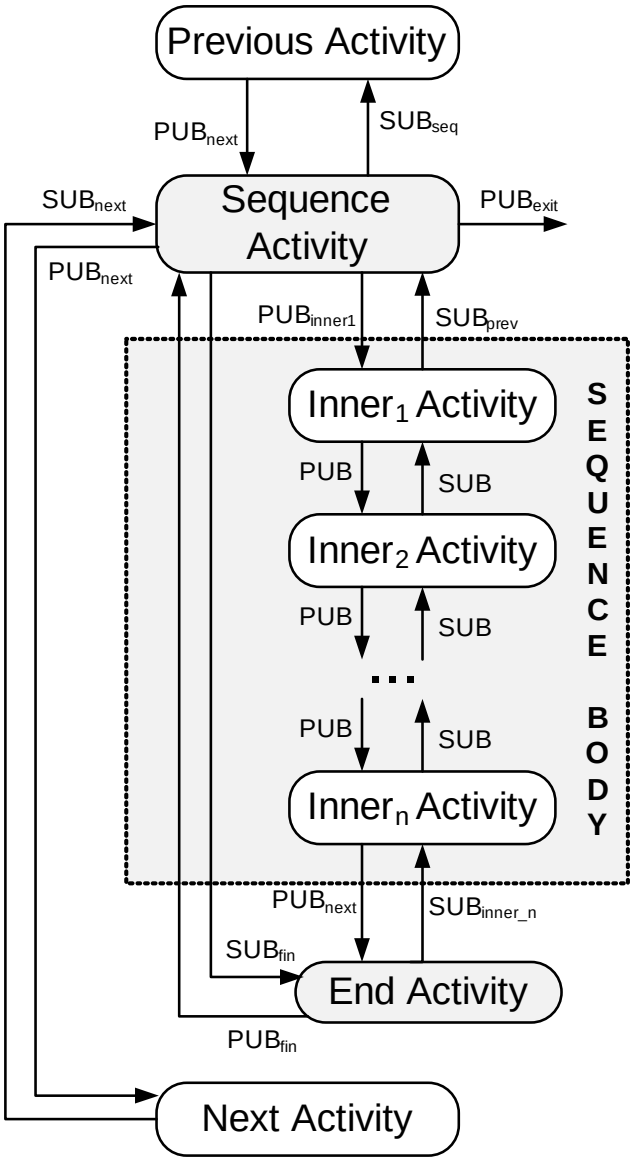


Figure A.9: Publish/Subscribe messages for the <sequence> activity.

inner activity finishes its execution. In this case the <sequence> agent receives a PUB_{fin} publication that represents the end of the body execution and publishes PUB_{next} that triggers the next activity in the BPEL process. If the operation fails, then a standard error publication is published.

A.2.2 <if> activity

The <if> activity implements the if statement in the BPEL language. Where if the expression condition is true the control passes to the corresponding execution path. This way the BPEL process can select exactly one activity for execution from a set of alternatives.

Listing A.18: BPEL syntax for the <if> activity.

```
<if standard-attributes>
    standard-elements
        <condition="bool-expr">
        activity
    <elseif>*
        <condition="bool-expr">
        activity
    </elseif>
    <else>?
        activity
    </else>
</if>
```

Listing A.19: Based on the student id the <if> activity assigns a student's surname to a student type.

```
<if condition="getVariableData('StudentId') & gt 100">
        <assign>
            <copy>
                <from variable="StudentData" part="surname" />
                <to variable="pdhStudent" part="surname" />
            </copy>
        </assign>
    <elseif condition="getVariableData('StudentId') &gt 50">
        <assign>
            <copy>
                <from variable="StudentData" part="surname" />
                <to variable="postgradStudent" part="surname" />
            </copy>
        </assign>
    </elseif>
    <else>
        <assign>
            <copy>
                <from variable="StudentData" part="surname" />
                <to variable="undergradStudent" part="surname" />
```

```
            </copy>
        </assign>
    </else>
</if>
```

<condition> activity subscription messages

SUB_{cnd}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

SUB_{fin}= [*class*, eq, END-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "EndActivityId"]

<condition> activity publication messages

PUB_{act_i}= publication to activity$_i$ that is the first activity in the body of the if, elseif, or else clauses

PUB_{next}= publication to next activity

<end> activity subscription messages

SUB_{end}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "LastInnerActivityId"]

<end> activity publication messages

PUB_{fin}= [*class*, END-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "EndActivityId"]

<if> activity subscription and publication algorithm

The worker responsible for the <condition> activity must subscribe to the activity that preceded the condition and to the activity <end>. The latter represents the final activity that is executed after one of the if, elseif, or else bodies finishes its execution. The first inner activity in the selected if, elseif, or else bodies subscribes to the condition activity. After the <condition> activity execution a publication is made to the next inner activity. The last activity in the if, elseif, or else bodies submit a publication that triggers the <end> activity.

Thus, the <condition> worker subscribes to SUB_{cnd} and waits for a proper trigger publication. After it is triggered, the worker evaluates the condition expression in each if, else, or elseif and for the first true condition sends a publication to the proper body activity. We must also notice that the <condition> agent also subscribes to the <end> (SUB_{fin}) and its triggered after the last inner activity finishes its execution. In this case

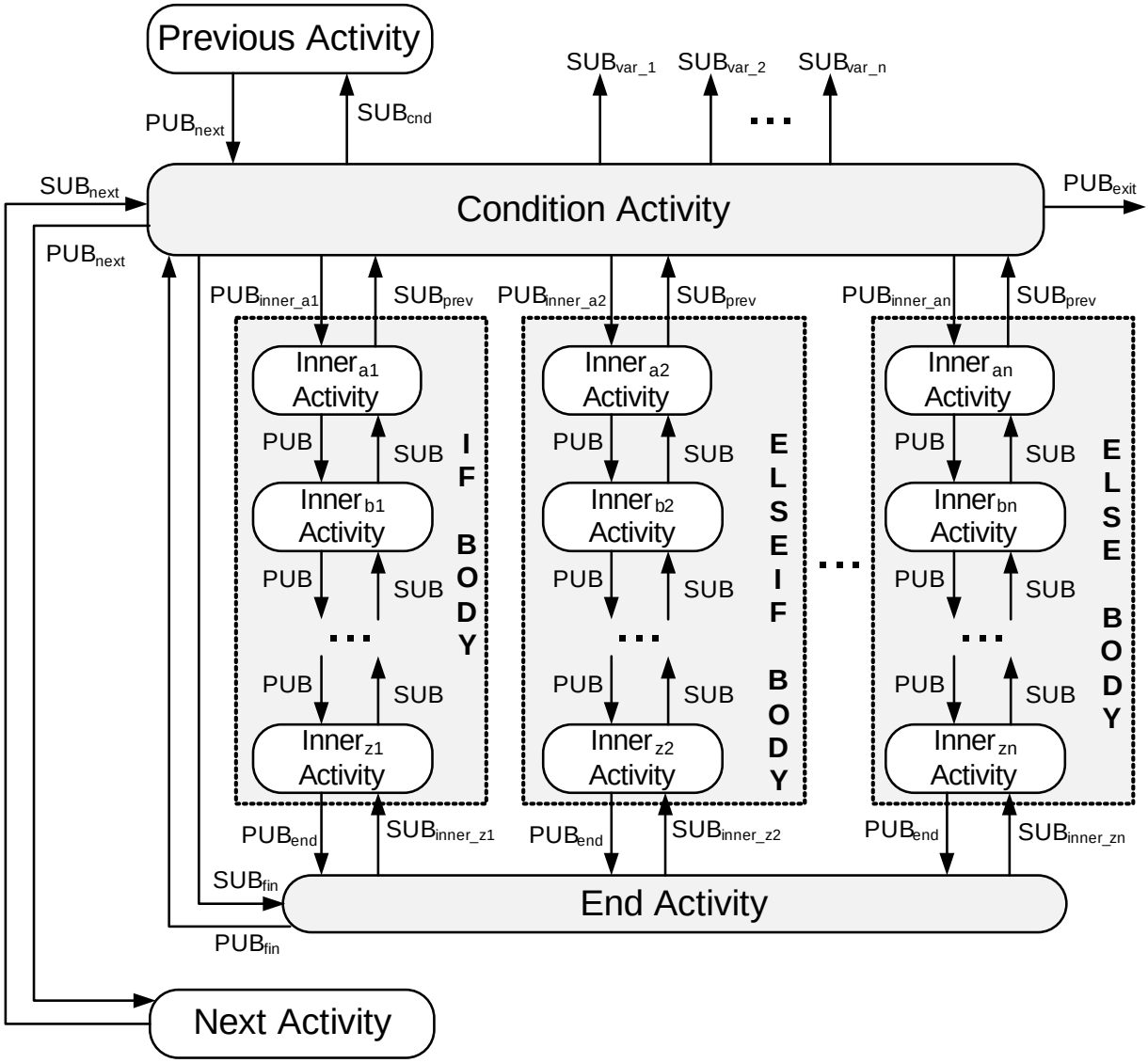


Figure A.10: Publish/Subscribe messages for the <if> activity.

Algorithm 25: <if> activity algorithm

subscribe to $\text{SUB}_{cnd} \vee \text{SUB}_{fin}$
if SUB_{cnd} was triggered **then**
  if <if> condition is true **then**
    publish PUB_{inner_1}
  else if <elseif$_1$> condition is true **then**
    publish PUB_{inner_2}
  else if <elseif$_2$> condition is true **then**
    ...
  else if <else> condition is true **then**
    publish PUB_{inner_n}
  end if
else if SUB_{fin} was triggered **then**
  publish PUB_{next}
end if

the <condition> agent receives a PUB_{fin} publication that represents the end of the body execution and publishes PUB_{next}, the latter triggers the next activity in the BPEL process. If the operation fails, then a standard error publication is published.

A.2.3 <while> activity

The <while> activity is used to define an iterative activity. The iterative activity is performed until the specified boolean condition no longer holds true.

Listing A.20: BPEL syntax for the <while> activity.

```
<while condition="bool-expr" standard-attributes>
    standard-elements
    activity+
</while>
```

Listing A.21: The example below presents a while loop; while the variableA is less than variable B a client web service operation is invoked.

```
<while condition=" getVariableData('variableA') &lt getVariableData('
   variableB')">
    <sequence>
    <invoke partnerLink="AmericanAirlines">
            portType="FlightAvailabilityPT"
            operation="FlightAvailability"
            inputVariable="variableC" />
    <assign name="assign1">
     <copy>
      <from expression="getVariableData('variableA')+1" />
      <to variable="variableA" />
```

```
     </copy>
    </assign>
    </assign>
    </sequence>
</while>
```

<while> activity subscription messages

SUB_{whl}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

SUB_{var_i}= [*class*, eq, VARIABLE-UPDATE], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "innerActivityId"],
[*variableName*, eq, "BPELVariableName"$_i$]

SUB_{fin}= [*class*, eq, END-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "EndActivityId"]

<while> activity publication messages

PUB_{inner_1}= publication to the first inner activity of the while loop

PUB_{next}= publication to next activity

<end> activity subscription messages

SUB_{end}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "LastInnerActivityId"]

<end> activity publication messages

PUB_{fin}= [*class*, END-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "EndActivityId"]

<while> activity subscription and publication algorithm

The worker responsible for the <while> activity must subscribe: a) to all the variables used into the while condition, b) to the activity that preceded the <while> activity, and c) to the activity <end>. The latter represents the final activity in the body of the <while>. Each activity in the body of the <while> subscribes to the previous inner activity and publish to the next inner activity, with two exceptions: a) the first inner activity subscribes to the <while> activity, and b) the last inner activity publishes to the <end> activity.

Thus, the <while> worker subscribes to SUB_{whl} and waits for a proper trigger publication. After it is triggered evaluates the while condition and either sends a publication

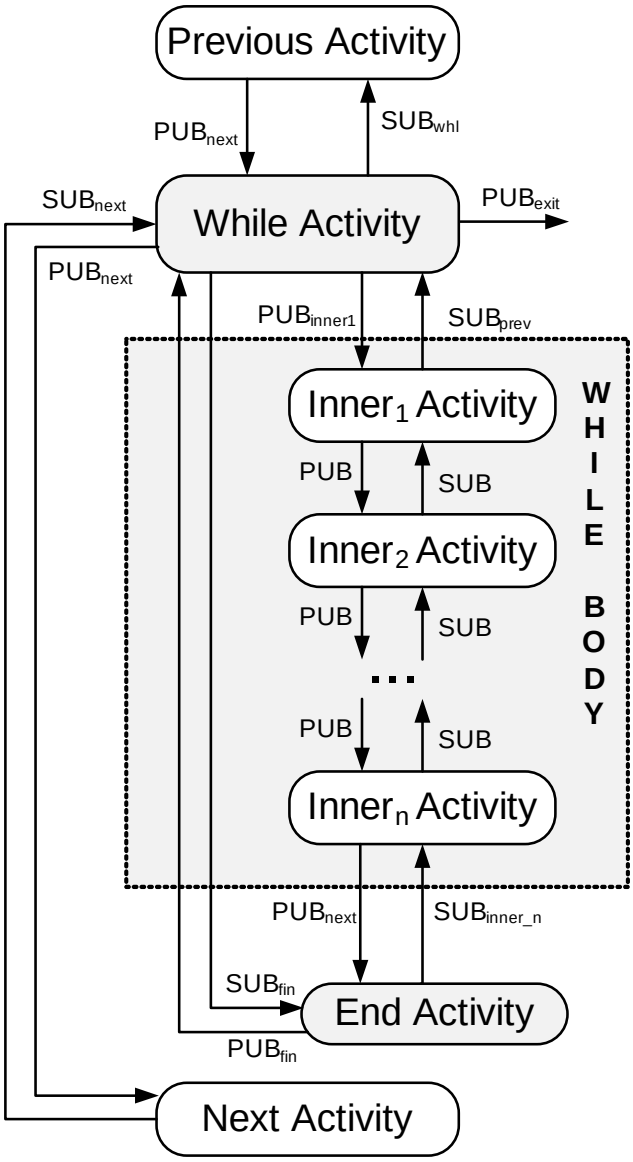


Figure A.11: Publish/Subscribe messages for the <while> activity.

Algorithm 26: <while> activity algorithm

subscribe to $\text{SUB}_{seq} \vee \text{SUB}_{fin}$
subscribe to all variables used by the while condition SUB_{var_i}
if $\text{SUB}_{seq} \vee \text{SUB}_{fin}$ was triggered **then**
  if while condition is true **then**
    publish PUB_{inner_1}
  else
    publish PUB_{next}
  end if
end if

to the first inner activity in the body of the while or sends a publication to the next activity. The body of the while is executed and the <end> activity is triggered after the last inner activity finishes its execution. In this case the <while> agent receives a PUB_{end} publication that represents the end of one iteration and causes the while condition to be evaluated again. If the condition is evaluated to false then the <while> agents publishes PUB_{next} that triggers the next activity in the BPEL process. If the operation fails, a standard error publication is published.

A.2.4 <pick> activity

The <pick> activity is used to wait for the occurrence of one of a set of events and then performs an activity associated with a message event or alarm event.

Listing A.22: BPEL syntax for the <pick> activity.

```
<pick createInstance="yes|no"? standard-attributes>
      standard-elements
    <onMessage partnerLink="NCName"
                  portType="QName"?
                  operation="NCName"
                  variable="BPELVariableName"?
                  messageExchange="NCName"?>+
      <correlations>?
            <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      activity
   </onMessage>
   <onAlarm>
   (for="duration-expr" | until="deadline-expr")>
    activity
   </onAlarm>*
</pick>
```

Listing A.23: <pick> example; if a client calls the operation ShutdownAlarm then the BPEL process stops the alarm. While if the timer expires after 20 minutes the alarm goes on.

```
<pick>
    <onMessage partnerLink="AmericanAirlines"
                portType="FlightCallbackPT"
                operation="FlightCallback"
                variable="FlightResponse">
        <!-- perform an activity -->
   </onMessage>

   <onAlarm for=" 'PT20M' ">
        <!-- perform an activity -->
   </onAlarm>

</pick>
```

<pick> activity subscription messages

SUB_{pick}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

SUB_{msg_i}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*partnerLink*, eq, "NCName"], [*portType*, eq, "QName"]?,
[*operation*, eq, "NCName"]

SUB_{var_i}= [*class*, eq, VARIABLE-UPDATE], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*variableName*, eq, "BPELVariableName"$_i$]

SUB_{fin}= [*class*, eq, END-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "EndActivityId"]

<pick> activity publication messages

PUB_{next}= publication to next activity

PUB_{onM}= [*class*, MSG-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "pickActivityId"]
[*targetActivityID*, "targetActivityId"]

PUB_{onA}= [*class*, ALARM-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "pickActivityId"]
[*targetActivityID*, "targetActivityId"]

<OnMessage> activity subscription messages

SUB_{onM}= [*class*, eq, MSG-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "pickActivityId"]
[*targetActivityID*, eq, "activityId"]

SUB_{var}= [*class*, eq, VARIABLE-UPDATE], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*variableName*, eq, "BPELVariableName"]

<OnMessage> activity publication messages

PUB_{inner}= publication to next inner activity

<OnAlarm> activity subscription messages

SUB_{onA}= [*class*, eq, ALARM-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "pickActivityId"]
[*targetActivityID*, eq, "activityId"]

<OnMessage> activity publication messages

PUB_{inner}= publication to next inner activity

<end> activity subscription messages

SUB_{end_i}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "LastInnerActivityId"$_i$]

<end> activity publication messages

PUB_{fin}= [*class*, END-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "EndActivityId"]

<pick> activity subscription and publication algorithm

The worker responsible for the <pick> activity must subscribe to the activity that preceded the <pick> (SUB_{pick}), to the activity <end> (SUB_{fin}), and to all messages that the activation of on message activities depends on (SUB_{msg_i}). The <end> activity represents the final activity in the body of the <pick>. There are two kind of activities in the body of the <pick>: a) OnAlarm activities and b) OnMessage activities. For each one there is a separate worker for its subscription evaluation and execution. Each activity in the body of the OnAlarm and OnMessage activities subscribes to the previous inner activity and publish to the next inner activity, with two exceptions: a) the first inner

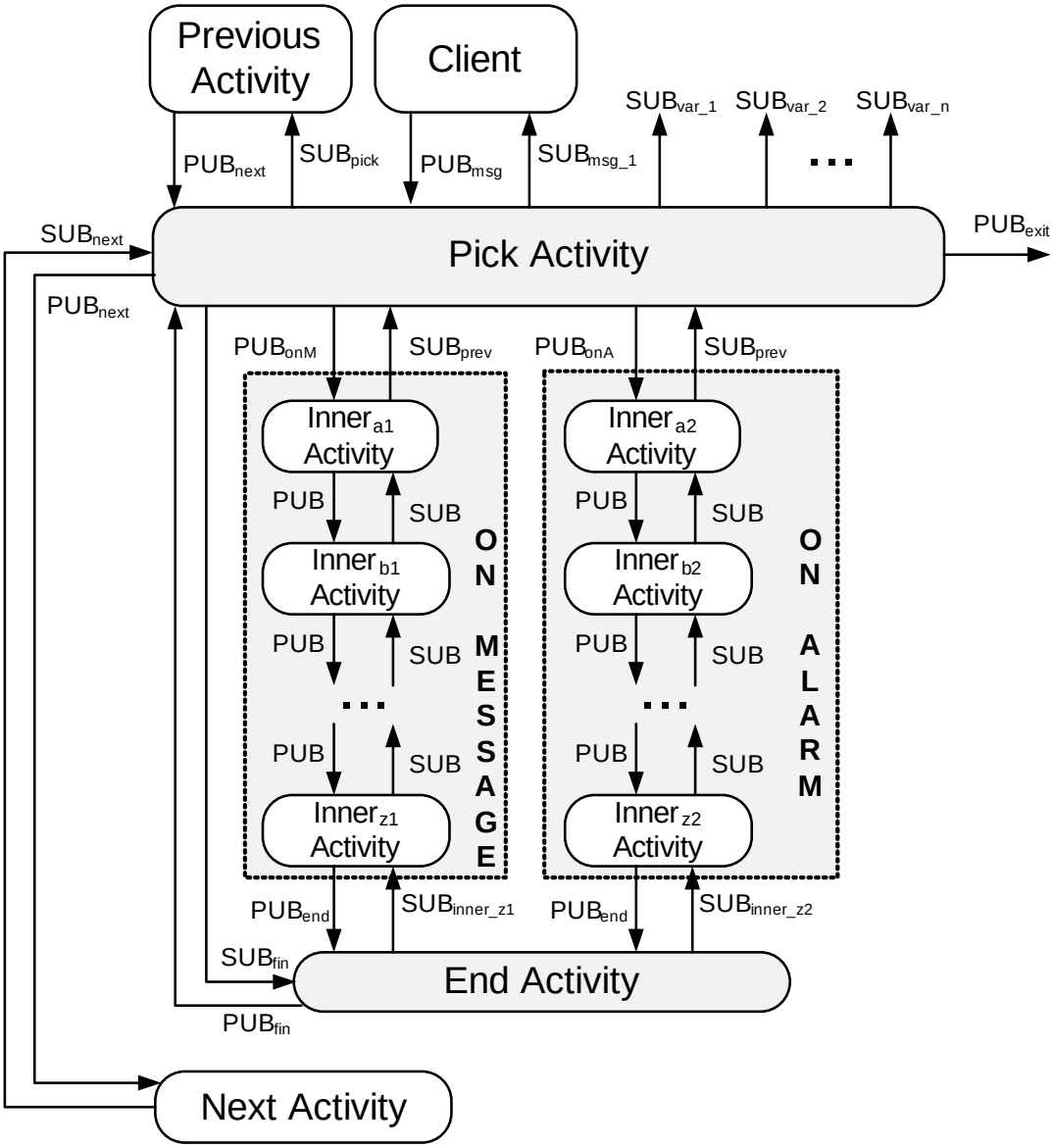


Figure A.12: Publish/Subscribe messages for the <pick> activity.

Algorithm 27: <pick> activity algorithm

```
subscribe to SUB_pick ∨ SUB_fin
if SUB_pick was triggered then
  while has not received any message or any alarm has expired do
    if SUB_msg_i triggered by message then
      publish PUB_onM
    end if
    if an alarm was triggered then
      publish PUB_onA
    end if
  end while
else if SUB_fin was triggered then
  publish PUB_next
end if
```

activity subscribes to the OnAlarm or OnMessage activity, and b) the last inner activity publishes to the <end> activity.

As already said the <pick> worker waits for a proper trigger publication. After it is triggered sets on its alarm timers and subscribes to the proper on message messages. If more than one event occurs, then the selection of the activity to be performed depends on which event occurred first and the agent sends the according publications to invoke the proper <OnMessage> or <OnAlarm> agent. When the <pick> agent receives a PUB_{fin} publication, the agent finishes its execution and sends a publication to the next activity. If the operation fails, a standard error publication is published.

We must note that the <pick> activity supports a special form to create an instance of a business process. In this case, no alarms are permitted and each onMessage is equivalent to a <receive> basic activity with the attribute "createInstance=yes".

<pick createInstance=yes> activity

This is a special form of <pick> than can be used to create an instance of the business process. In this case, no alarms are permitted and each onMessage is equivalent to a <receive> activity with the attribute createInstance=yes.

<pick createInstance=yes> activity subscription messages

SUB_{msg_i}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*activityID*, eq, "activityId"], [*variableName*, eq, "BPELVariableName"],
[*partnerLink*, eq, "NCName"], [*portType*, eq, "QName"]?,
[*operation*,eq, "NCName"]

SUB_{var_i}= [*class*, eq, VARIABLE-UPDATE], [*process*, eq, "processName"],

[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"],
[*variableName*, eq, "BPELVariableName"$_i$]

<pick createInstance=yes> activity publication messages

PUB$_{inst}$= [*class*, INSTANCE], [*process*, "processName"], <<instanceId>>

PUB$_{var}$= [*class*, VARIABLE_UPDATE], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "activityId"],
[*variableName*, "BPELVariableName"], <<variableData>>

PUB$_{next}$= publication to the next activity

<pick createInstance=yes> activity subscription and publication algorithm

Algorithm 28: <pick createInstance=yes> activity algorithm

```
if createInstance == yes then
    subscribe to all SUB_msg_i and SUB_var_i
    if SUB_msg_k was triggered then
        publish {PUB_inst, PUB_var, PUB_next}
    end if
end if
```

If the <pick createInstance=yes> activity has the create instance element set to *yes* then the worker agent must subscribe to all SUB$_{msg_i}$ and SUB$_{var_i}$. After it is triggered with the appropriate publications the worker agent executes the associated receive operation. If the <pick createInstance=yes> is successfully executed, the worker publishes PUB$_{var}$, PUB$_{next}$, and PUB$_{inst}$. Finally, when the operation fails, a standard error publication is published.

A.2.5 <flow> activity

The <flow> activity implements the concurrent execution of enclosed activities and provides links that can synchronize their execution.

Listing A.24: BPEL syntax for the <flow> activity.

```
<flow standard-attributes>
    standard-elements
    <links>?
        <link name="NCName">+
    </links>
    activity+
</flow>
```

Listing A.25: <flow> example; parallel execution of two <sequence> activities. The first one contains an <invoke> and a <receive> activity. The second contains an <invoke> and a <pick> activity.

```
<flow name="FlowExample">
    <sequence>
        <invoke name="A1" .../>
        <receive name="B1" .../>
    </sequence>
    <sequence>
        <invoke name="A2" .../>
        <pick name="B2" .../>
    </sequence>
</flow>
```

<flow> activity subscription messages

SUB$_{flow}$= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

SUB$_{fin}$= [*class*, eq, END-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "EndactivityId"]

<flow> activity publication messages

PUB$_{inner_i}$= [*class*, FLOW-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "flowActivityId"]

PUB$_{next}$= publication to the next activity

CASE 1: First activity in the flow and not target of any link (in example activities A_1, A_2)

activity subscription messages

SUB$_{prev}$= [*class*, eq, FLOW-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "flowActivityId"]

activity publication messages

PUB$_{next}$= [*class*, FLOW-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "activityId"]

CASE 2: activity is not target to any link, is not the the first activity in the flow, and there is no any target of link in its order supplier activity (activities B_2, C_2)

activity subscription messages

SUB$_{prev}$= [*class*, eq, FLOW-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

activity publication messages

PUB$_{next}$= [*class*, FLOW-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "activityId"]

CASE 3: activity is target of links (activities B_1)

When a activity that acts as a source of a link completes, its corresponding agent determines the status its outgoing links. To determine the status for each link, the transitionCondition is evaluated. If the evaluation is true, then the status is positive, otherwise the status is negative. Thus activity agent publishes the PUB$_{link}$, after the evaluation of the transitionCondition.

activity publication messages

PUB$_{link}$= [*class*, LINK-STATUS], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "activityId"],
<<status , "POSITIVE|NEGATIVE">>

For each activity that has a synchronization dependency on the source activity, its corresponding agent subscribes to the execution status of its order supplier activities and the status of all incoming links. For example the agent for activity B_1 subscribes to the execution status of A_1 and the incoming link status of A_2.

activity subscription messages

SUB$_{prev}$= [*class*, eq, FLOW-ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

SUB$_{link}$= [*class*, eq, LINK-STATUS], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "linkActivityId"]

CASE 4: activity which is not target of a link but there is some target of link as its order supplier activity such as activity C_1

activity subscription messages

SUB$_{prev}$= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "prevActivityId"]

<end> activity subscription messages

SUB_{end_i}= [*class*, eq, ACTIVITY], [*process*, eq, "processName"],
[*instanceID*, isPresent, "instanceId"], [*activityID*, eq, "LastInnerActivityId"$_i$]

<end> activity publication messages

PUB_{end}= [*class*, END-ACTIVITY], [*process*, "processName"],
[*instanceID*, "instanceId"], [*activityID*, "EndActivityId"]

<flow> activity subscription and publication algorithm

Algorithm 29: <flow> activity algorithm

```
subscribe to SUB_flow ∨ SUB_fin
if SUB_flow was triggered then
  publish PUB_inner
else if SUB_fin was triggered then
  publish PUB_next
end if
```

The worker responsible for the <flow> activity must subscribe to the activity that preceded the <flow> and to the activity <end>. The latter represents the final activity in the body of the parallel branch that is executed. There are four kind of activities in the body of the flow: a) initial activities that are not targets of any links (like the activities A_1, A_2), b) activities that are not target to any link, are not initial activities, and they have no target of any link in their order supplier activities (like activities B_2, C_2), c) activities that are target of links (like activity B_1), and d) activities which are not target of any link by themselves but exists an activity that is target of a link in their order supplier activities (such as activity C_1).

For each <flow> inner activity there is a separate worker responsible for its subscription evaluation and execution. Based on the category that belongs the worker may subscribe to the execution or link status of its previous activities.

The <end> activity subscribes to the last inner activity of each parallel flow and waits for all parallel execution paths to finish. Otherwise an error is published after a period of time. In this case the <flow> agent receives a PUB_{end} publication, finishes its execution and sends a publication to the next activity. If any activity operation fails, a standard error publication is published.

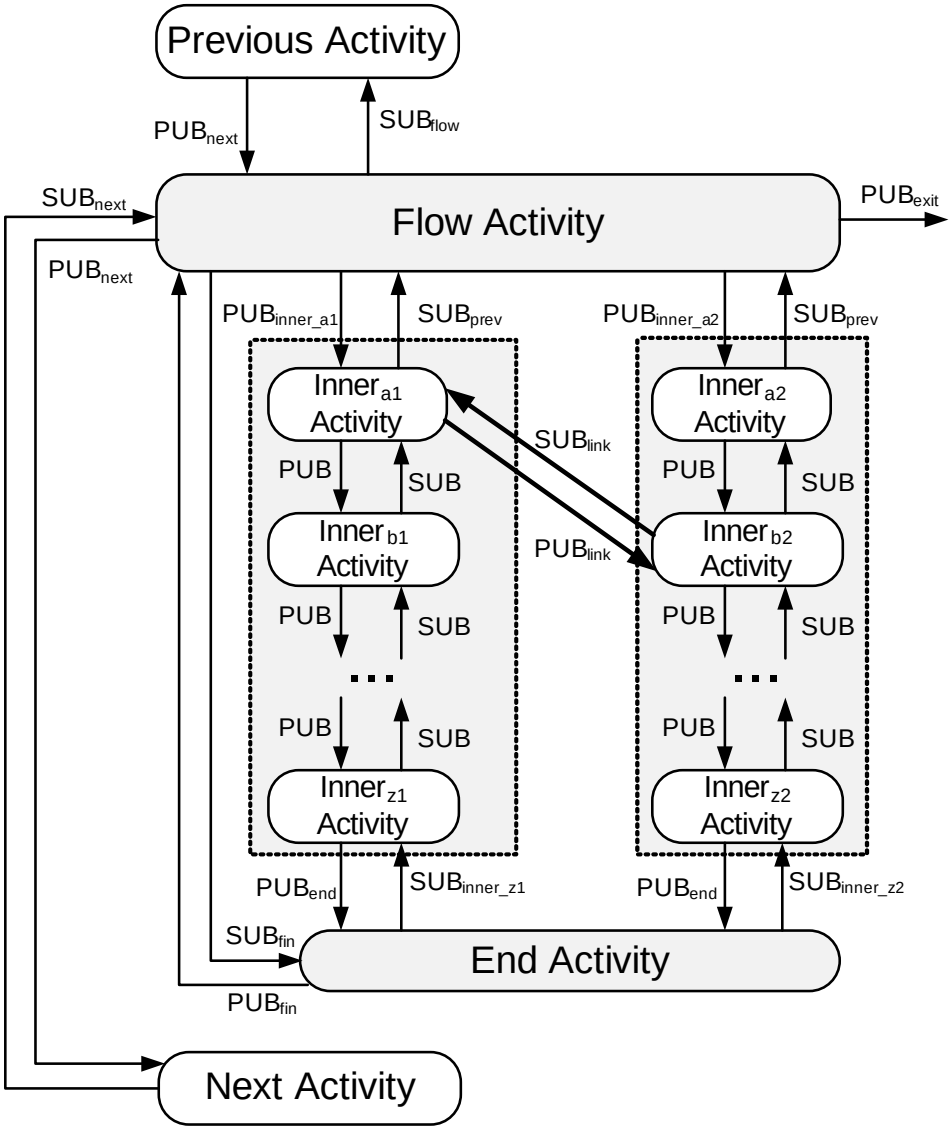


Figure A.13: Publish/Subscribe messages for the <flow> activity.

Acronyms

API	Application Programming Interface
BISON	Biology Inspired techniques for Self Organization in dynamic Networks
BPEL	Business Process Execution Language
CAN	Content Addressable Network
CFG	Control Flow Graph
CORBA	Common Object Request Broker Architecture
DELIS	Dynamically Evolving Large-scale Information Systems
DHT	Distributed Hash Table
EU	European Union
FTP	File Transfer Protocol
GPL	GNU General Public License
HTTP	Hypertext Transfer Protocol
IT	Information Technology
JMS	Java Message Service
LCC	Lightweight Coordination Calculus
MIME	Multipurpose Internet Mail Extensions
OASIS	Organization for the Advancement of Structured Information Standards
OSI	Open Systems Interconnection
P2P	Peer-to-Peer
PC	Personal Computer
PDG	Program Dependence Graph
Pub/Sub	Publish/Subscribe
Qos	Quality of Service
REST	Representational State Transfer
RMI	Remote Method Invocation
RP	Rendezvous Point
SHA	Secure Hash Algorithm
SMTP	Simple Mail Transfer Protocol
SOA	Service-Oriented Architecture

SOAP	Simple Object Access Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TTL	Time To Live
UDDI	Universal Description, Discovery and Integration
WfMS	Workflow Management System
WS	Web Service
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Services Description Language
XML	Extensible Markup Language
XPath	XML Path Language

Bibliography

[AAA+07] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0. Technical report, OASIS Standard, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services specification, version 1.1. Technical report, Specification BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems, 2003. http://public.dhe.ibm.com/software/dw/specs/ws-bpel/ws-bpel1.pdf.

[Act11] ActiveVOS. Activevos product overview. http://www.activevos.com/products/activevos/overview, May 2011.

[AHKB03] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14:5–51, July 2003.

[ASS+99] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC '99, pages 53–61, New York, NY, USA, 1999. ACM.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36:335–371, December 2004.

[BB05] Stefan Birrer and Fabian E. Bustamante. The feasibility of dht-based streaming multicast. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecom-*

munication Systems, pages 288–298, Washington, DC, USA, 2005. IEEE Computer Society.

[BCD+05] Ozalp Babaoglu, Geoffrey Canright, Andreas Deutsch, Gianni Di Caro, Frederick Ducatelle, Luca Gambardella, Niloy Ganguly, Márk Jelasity, Roberto Montemanni, and Alberto Montresor. Design patterns from biology for distributed computing. In *Proceedings of the European Conference on Complex Systems*, November 2005.

[BCM+99] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 262 –272, 1999.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1. Technical report, W3C Note, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[BHM+01] Jean Bacon, Alexis Hombrecher, Chaoying Ma, Ken Moody, and Walt Yao. Event storage and federation using odmg. In *Revised Papers from the 9th International Workshop on Persistent Object Systems*, POS-9, pages 265–281, London, UK, 2001. Springer-Verlag.

[BHM+04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. Technical report, W3C Working Group Note, 2004. http://www.w3.org/TR/ws-arch/.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36:37–53, December 1993.

[BMM05] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Workflow partitioning in mobile information systems. In Elaine Lawrence, Barbara Pernici, and John Krogstie, editors, *Mobile Information Systems*, volume 158 of *IFIP International Federation for Information Processing*, pages 93–106. Springer Boston, 2005. 10.1007/0-387-22874-8_7.

[BPSM+06] T. Bray, J. Paol, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible markup language (xml) 1.1. Technical report, W3C Recommendation, 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/.

[BV04] Paul Buhler and José M. Vidal. Enacting BPEL4WS specified workflows with multiagent systems. In *Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

[CCMN04] Girish B. Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 134–143, New York, NY, USA, 2004. ACM.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. Technical report, W3C Note, 2001. http://www.w3.org/TR/wsdl.

[CCR04] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 9–, Washington, DC, USA, 2004. IEEE Computer Society.

[CDK+03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 298–313, New York, NY, USA, 2003. ACM.

[CDNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27:827–850, September 2001.

[CdSLMM11] A.P. Couto da Silva, E. Leonardi, M. Mellia, and M. Meo. Chunk distribution in mesh-based large-scale p2p streaming systems: A fluid approach. *Parallel and Distributed Systems, IEEE Transactions on*, 22(3):451 –463, march 2011.

[CHvRR04] L. Clement, A. Hately, C. von Riegen, and T. Rogers. Uddi version 3.0.2. Technical report, UDDI Spec Technical Committee Draft, 2004. http://uddi.org/pubs/uddi_v3.htm.

[Coa99] Workflow Management Coalition. The workflow management coalition specification: Terminology and glossary. Technical report, Workflow Management Coalition, 1999. http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19:332–383, August 2001.

[CRW04] A. Carzaniga, M.J. Rutherford, and A.L. Wolf. A routing scheme for content-based networking. In *INFOCOM 2004. Twenty-third AnnualJoint*

Conference of the IEEE Computer and Communications Societies, volume 2, pages 918 – 928 vol.2, March 2004.

[CS02] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '02, pages 177–190, New York, NY, USA, 2002. ACM.

[CS05] F. Cao and J.P. Singh. Medym: match-early and dynamic multicast for content-based publish-subscribe service networks. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pages 370 – 376, June 2005.

[CW03] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 163–174, New York, NY, USA, 2003. ACM.

[del06] delis. DELIS: Dynamically Evolving, Large-scale Information Systems. http://delis.upb.de, 2006.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM.

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 254–269, New York, NY, USA, 2001. ACM.

[EJ01] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001. Updated by RFC 4634.

[FGM+97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[FPR02] Geoffrey Fox, Shrideep Pallickara, and Xi Rao. A scaleable event infrastructure for peer to peer grids. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pages 66–75, New York, NY, USA, 2002. ACM.

[GDS+03] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 314–329, New York, NY, USA, 2003. ACM.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77, January 1983.

[GMS06] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks: Algorithms and evaluation. *Performance Evaluation*, 63(3):241 – 263, 2006. P2P Computing Systems.

[Gnu] Gnutella. The gnutella protocol specification v0.4. http://www.stanford.edu/class/cs244b/gnutella_protocol_0.4.pdf.

[GRCB05] Li Guo, David Robertson, and Yun-Heh Chen-Burger. A novel approach for enacting the distributed business workflows using bpel4ws on the multi-agent platform. In *Proceedings of the IEEE International Conference on e-Business Engineering*, pages 657–664, Washington, DC, USA, 2005. IEEE Computer Society.

[GSAA04] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[GSG02] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 5–18, New York, NY, USA, 2002. ACM.

[HFC+08] Yan Huang, Tom Z.J. Fu, Dah-Ming Chiu, John C.S. Lui, and Cheng Huang. Challenges, design and analysis of a large-scale p2p-vod system. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 375–388, New York, NY, USA, 2008. ACM.

[HH06] Dominic Heutelbeck and Matthias Hemmje. Distributed leader election in p2p systems for dynamic sets. *Mobile Data Management, IEEE International Conference on*, 0:29, 2006.

[HHL+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, pages 321–332. VLDB Endowment, 2003.

[Hol95] D. Hollingsworthd. The workflow management coalition specification: The workflow reference model. Technical report, Workflow Management Coalition, 1995. http://www.wfmc.org/Articles-White-Papers/.

[HX07] Seung Chul Han and Ye Xia. Optimal leader election scheme for peer-to-peer applications. *International Conference on Networking*, 0:29, 2007.

[IBM11] IBM. Websphere. http://www-01.ibm.com/software/websphere/, May 2011.

[jBo11] jBoss. jbpm5. http://www.jboss.org/jbpm, May 2011.

[Jen87] Kurt Jensen. Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin / Heidelberg, 1987. 10.1007/BFb0046842.

[JPPnMMJ08] Ricardo Jiménez-Peris, Marta Patiño Martínez, and Ernestina Martel-Jordán. Decentralized web service orchestration: a reflective approach. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 494–498, New York, NY, USA, 2008. ACM.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 84–95, New York, NY, USA, 2002. ACM.

[LJ05] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, pages 249–269, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[LMJ10] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web*, 4:2:1–2:33, January 2010.

[MÖ1] Gero Mühl. Generic constraints for content-based publish/subscribe. In *Proceedings of the 9th International Conference on Cooperative Information Systems*, CoopIS '01, pages 211–225, London, UK, 2001. Springer-Verlag.

[MÖ2] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, University of Darmstad, 2002.

[MB02] Alberto Montresor and Ozalp Babaoglu. The BISON project. *IEEE Computational Intelligence Bulletin*, 1(1):6–9, December 2002.

[MJ09] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, 2009.

[MPR+03] Alan Mislove, Ansley Post, Charles Reis, Paul Willmann, Peter Druschel, Dan S. Wallach, Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, and Luciana Arantes-Bezerra. Post: a secure, resilient, cooperative messaging system. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, pages 11–11, Berkeley, CA, USA, 2003. USENIX Association.

[Nap] LLC Napster. www.napster.com/.

[NCS04] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing execution of composite web services. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 170–187, New York, NY, USA, 2004. ACM.

[NWD03] Tsuen-Wan Ngan, Dan Wallach, and Peter Druschel. Enforcing fair sharing of peer-to-peer resources. In *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 149–159. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-45172-3_14.

[OAA+00] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms*, Middleware '00, pages 185–207, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[ODE11] Apache ODE. Apache ode (orchestration director engine). http://ode.apache.org/, May 2011.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19:177–184, April 1984.

[OPSS93] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 58–68, New York, NY, USA, 1993. ACM.

[PB02] P.R. Pietzuch and J.M. Bacon. Hermes: a distributed event-based middleware architecture. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 611 – 618, 2002.

[PFL+00] João Pereira, Françoise Fabret, François Llirbat, Radu Preotiuc-Pietro, Kenneth A. Ross, and Dennis Shasha. Publish/subscribe on the web at extreme speed. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 627–630, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[PFLS00] João Pereira, Françoise Fabret, François Llirbat, and Dennis Shasha. Efficient matching for web-based publish/subscribe systems. In *Proceedings of the 7th International Conference on Cooperative Information Systems*, CooplS '02, pages 162–173, London, UK, 2000. Springer-Verlag.

[PPC97] Santanu Paul, Edwin Park, and Jarir Chaar. Rainman: a workflow system for the internet. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 15–15, Berkeley, CA, USA, 1997. USENIX Association.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 311–320, New York, NY, USA, 1997. ACM.

[PWR04] Ginger Perng, Chenxi Wang, and Michael K. Reiter. Providing content-based services in a peer-to-peer environment. In *in Proceedings of the third International Workshop on Distributed Event-Based Systems (DEBS*, pages 74–79, 2004.

[RB11] Marco Mellia Arpad Bakay Tivadar Szemethy Fabien Mathieu Luca Muscariello Saverio Niccolini Jan Seedorf Giuseppe Tropea Robert Birke, Emilio Leonardi. Architecture of a network-aware p2p-tv application: the napa-wine approach. *IEEE Communication Magazine*, 2011.

[RD01a] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 188–201, New York, NY, USA, 2001. ACM.

[RD01b] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, 2001. Springer-Verlag.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[RKCD01] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, NGC '01, pages 30–43, London, UK, 2001. Springer-Verlag.

[RS04] Venugopalan Ramasubramanian and Emin Gün Sirer. The design and implementation of a next generation name service for the internet. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 331–342, New York, NY, USA, 2004. ACM.

[RSW97] F. Ranno, S. K. Shrivastava, and S. M. Wheater. A system for specifying and coordinating the execution of reliable distributed aplications. Technical report, University of Bologna, 1997.

[SA97] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceddings AUUG Technical Conference*, September 1997.

[Ser11] BizTalk Server. Biztalk server 2010. http://www.microsoft.com/biztalk/en/us/default.aspx, May 2011.

[SMK08] Emil Sit, Robert Morris, and M. Frans Kaashoek. Usenetdht: a low-overhead design for usenet. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 133–146, Berkeley, CA, USA, 2008. USENIX Association.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11:17–32, February 2003.

[SPPvS08] Michal Szymaniak, David Presotto, Guillaume Pierre, and Maarten van Steen. Practical large-scale latency estimation. *Comput. Netw.*, 52:1343–1364, May 2008.

[SPvS04] M. Szymaniak, G. Pierre, and M. van Steen. Scalable cooperative latency estimation. In *Parallel and Distributed Systems, 2004. ICPADS 2004.*

Proceedings. Tenth International Conference on, pages 367 – 376, july 2004.

[SS05] Kundan Singh and Henning Schulzrinne. Peer-to-peer internet telephony using sip. In *Proceedings of the international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '05, pages 63–68. ACM, 2005.

[TA04] Peter Triantafillou and Ioannis Aekaterinidis. Content-based publish/-subscribe over structured p2p networks. In *Proc. third Int. Workshop Distributed Event-based Systems (DEBS'04), 16 of 16 R. BALDONI et al*, pages 24–25, 2004.

[TAaJ03] David Tam, Reza Azimi, and Hans arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *Proceedings of the 1st International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, pages 138–152, 2003.

[Tar07] Sasu Tarkoma. Dynamic filter merging for publish/subscribe. In *World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a*, pages 1 –9, june 2007.

[TBF+03] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, DEBS '03, pages 1–8, New York, NY, USA, 2003. ACM.

[TE04] P. Triantafillou and A. Economides. Subscription summarization: a new paradigm for efficient publish/subscribe systems. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 562 – 571, 2004.

[TKLB07] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 49–60, New York, NY, USA, 2007. ACM.

[Vin97] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46 –55, feb 1997.

[VKK07] R. Vijayprasanth, R. Kavithaa, and Rajkumar Kettimuthu. Legs: A wsrf service to estimate latency between arbitrary hosts on the internet. In

Parallel and Distributed Processing Techniques and Applications, pages 360–364, 2007.

[WG08] Bernard Wong and Saikat Guha. Quasar: a probabilistic publish-subscribe system for social networks. In *Proceedings of the 7th international conference on Peer-to-peer systems*, IPTPS'08, pages 2–2, Berkeley, CA, USA, 2008. USENIX Association.

[YCP04] K. Park Y. Choi and D. Park. Homed: A peer-to-peer overlay architecture for large-scale content-based publish/subscribe systems. In *in Proceedings of the third International Workshop on Distributed Event-Based Systems (DEBS*, pages 20–25, 2004.

[YG07] Ustun Yildiz and Claude Godart. Towards decentralized service orchestrations. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1662–1666, New York, NY, USA, 2007. ACM.

[YGM94] Tak W. Yan and Héctor García-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19:332–364, June 1994.

[YZH07a] Xiaoyu Yang, Yingwu Zhu, and Yiming Hu. A large-scale and decentralized infrastructure for content-based publish/subscribe services. *Parallel Processing, International Conference on*, 0:61, 2007.

[YZH07b] Xiaoyu Yang, Yingwu Zhu, and Yiming Hu. Scalable content-based publish/subscribe services over structured peer-to-peer networks. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '07, pages 171–178, Washington, DC, USA, 2007. IEEE Computer Society.

[ZH05] Y. Zhu and Y. Hu. Ferry: an architecture for content-based publish/subscribe services on p2p networks. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 427 – 434, June 2005.

[ZHS+04] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41 – 53, January 2004.

[ZZJ+01] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '01, pages 11–20, New York, NY, USA, 2001. ACM.