Flexible Distributed Business Process Management

by

Vinod Muthusamy

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2011 by Vinod Muthusamy
Abstract

Flexible Distributed Business Process Management

Vinod Muthusamy

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2011

Many large business processes are inherently distributed, spanning multiple organizations, administrative domains, and geographic locations. To support such applications, this thesis develops a flexible and distributed platform to develop, execute, and monitor business processes. The solutions utilize a distributed content-based publish/subscribe overlay that is extended with support for mobile clients and client interest churn. Over this layer, a distributed execution engine uses events to coordinate the execution of the process, and dynamically redeploy activities in the process in order to minimize a user-specified cost function and preserve service level agreements (SLAs). Finally, a management layer allows users to find and automatically compose services available across a distributed set of service registries, and monitor processes for SLA violations. Evaluations show that the distributed execution engine can scale better than alternate architectures, exhibiting over 60% improvements in execution time in one experiment. As well the system can dynamically redeploy processes to reflect changing workload conditions and SLAs, saving up to 90% of the process messaging overhead of a static deployment.
## Contents

1 Introduction .......................................................... 1  
1.1 Motivation .......................................................... 1  
1.2 Problem statement .................................................. 7  
1.2.1 Dynamic publish/subscribe client interactions ............... 7  
1.2.2 Distributed process execution .................................. 10  
1.2.3 Service management and monitoring .......................... 12  
1.3 Contributions ....................................................... 15  
1.4 Organization ......................................................... 18  

2 Background and related work ........................................ 19  
2.1 Content-based publish/subscribe .................................. 19  
2.1.1 Matching semantics ............................................. 19  
2.1.2 Content-based routing .......................................... 21  
2.1.3 PADRES broker .................................................. 22  
2.1.4 Covering and merging .......................................... 23  
2.1.5 Client mobility .................................................. 25  
2.1.6 Congestion control .............................................. 26  
2.2 Business processes and workflows ............................... 26  
2.2.1 SOA development cycle ........................................ 26  
2.2.2 Business Process Execution Language ........................ 28  
2.2.3 Distributed workflows ......................................... 29  
2.2.4 Stream processing .............................................. 31  
2.2.5 Dynamic redeployment ......................................... 32  
2.3 Service management and monitoring ............................. 33  
2.3.1 Service and resource discovery ................................ 33  
2.3.2 Service composition ............................................ 36  
2.3.3 SLA modeling and monitoring ................................. 42  

3 Transactional client movement ....................................... 45  
3.1 Transaction properties ............................................. 45  
3.1.1 Movement operation ............................................ 46  
3.1.2 Publish/Subscribe system layers .............................. 47  
3.1.3 Client layer ...................................................... 48  
3.1.4 Notifications ..................................................... 49  
3.1.5 Routing layer .................................................... 50
3.2 Client movement protocol ........................................... 52
  3.2.1 System model .................................................. 52
  3.2.2 Client layer .................................................. 53
  3.2.3 Notifications .................................................. 56
  3.2.4 Routing layer .................................................. 57
3.3 Evaluation ............................................................... 60

4 Incremental filter aggregation ........................................ 70
  4.1 Subscription propagation properties ................................. 70
    4.1.1 Nature of subscription propagation .......................... 71
    4.1.2 Subscription similarity ................................... 73
    4.1.3 Non-decreasing similarity .................................. 75
  4.2 Subscription packing ............................................... 76
    4.2.1 Traditional subscription relation graph ...................... 76
    4.2.2 Simplified subscription relation graph ...................... 78
    4.2.3 Subscription packing algorithm ............................. 80
  4.3 Incremental subscription tree pruning ............................. 81
    4.3.1 Statistics collection ...................................... 82
    4.3.2 Subscription tree pruning .................................. 83
  4.4 Evaluation ........................................................... 84
    4.4.1 Methodology ................................................. 85
    4.4.2 Effect of covering ......................................... 88
    4.4.3 Comparisons among algorithms .............................. 89
    4.4.4 Varying number of unsubscriptions ......................... 92
    4.4.5 Varying subscription covering degree ....................... 93
    4.4.6 Varying similarity threshold $\theta$ ......................... 95
    4.4.7 Distributed workflow scenario ............................. 96
    4.4.8 Complex publish/subscribe matching engine ................. 97
    4.4.9 Clustered placement ....................................... 100
    4.4.10 Mobile scenario .......................................... 101

5 Distributed execution engine ......................................... 105
  5.1 Distributed process execution ................................... 105
    5.1.1 Execution architectures ................................... 105
    5.1.2 NIÑOS system architecture ................................ 107
    5.1.3 Process transformation .................................... 110
    5.1.4 Process deployment ........................................ 117
    5.1.5 Process execution .......................................... 120
    5.1.6 Process management ........................................ 121
    5.1.7 Example .................................................... 124
    5.1.8 Architectural benefits ..................................... 127
  5.2 Evaluation ........................................................... 130
    5.2.1 Experimental setup ........................................ 130
    5.2.2 Request rate ............................................... 131
    5.2.3 Web service delay .......................................... 133
8.2 Evaluation ................................................................. 202
  8.2.1 Setup ............................................................... 203
  8.2.2 Results ............................................................ 204

9 SLA modeling and monitoring 212
  9.1 SLA model ............................................................ 212
     9.1.1 Scopes .......................................................... 214
     9.1.2 Metrics .......................................................... 215
     9.1.3 Metric type example ........................................... 217
     9.1.4 Service level objectives ....................................... 219
     9.1.5 Action handlers ............................................... 220
     9.1.6 Service level agreements ..................................... 220
     9.1.7 Discussion .................................................... 221
     9.1.8 Implementation ............................................... 222
  9.2 SLA monitoring ..................................................... 223
     9.2.1 Validation ..................................................... 223
     9.2.2 Event enabling ............................................... 224
     9.2.3 SLA execution ............................................... 224
  9.3 Case study ......................................................... 226
     9.3.1 Modeling the SLAs ........................................... 228
     9.3.2 Validation .................................................... 231
     9.3.3 Generation of monitor rules ................................ 232

10 Conclusions 233
  10.1 Summary .......................................................... 233
  10.2 Future work ...................................................... 238

Bibliography 240
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Examples of subscriptions, advertisements and intersection relations</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Summary of BPEL activities</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Portion of the database at Broker I in Fig. 4.5</td>
<td>83</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of total broker processing time (in seconds) for different message types under active covering, lazy covering, and no covering algorithms. (The average processing time per message (in milliseconds) is shown in parentheses.)</td>
<td>88</td>
</tr>
<tr>
<td>4.3</td>
<td>Varying similarity threshold $\theta$</td>
<td>95</td>
</tr>
<tr>
<td>5.1</td>
<td>Agent messages for loan application process in Figure 5.11</td>
<td>127</td>
</tr>
<tr>
<td>6.1</td>
<td>Cost model components</td>
<td>140</td>
</tr>
<tr>
<td>6.2</td>
<td>Examples of cost functions</td>
<td>141</td>
</tr>
<tr>
<td>7.1</td>
<td>Supported models</td>
<td>166</td>
</tr>
<tr>
<td>8.1</td>
<td>Subscriptions and advertisements for services in Figure 8.2</td>
<td>193</td>
</tr>
<tr>
<td>9.1</td>
<td>Metric type for measuring number of invocations</td>
<td>217</td>
</tr>
<tr>
<td>9.2</td>
<td>Metric for measuring number of local accesses</td>
<td>218</td>
</tr>
<tr>
<td>9.3</td>
<td>Metric for measuring number of remote accesses</td>
<td>218</td>
</tr>
<tr>
<td>9.4</td>
<td>Metric for computing percentage</td>
<td>218</td>
</tr>
<tr>
<td>9.5</td>
<td>Metric for computing percentage of remote accesses</td>
<td>219</td>
</tr>
<tr>
<td>9.6</td>
<td>Example of SLO type</td>
<td>219</td>
</tr>
<tr>
<td>9.7</td>
<td>Example of SLO</td>
<td>219</td>
</tr>
<tr>
<td>9.8</td>
<td>Metric type for measuring execution time</td>
<td>228</td>
</tr>
<tr>
<td>9.9</td>
<td>Metric for measuring total execution time</td>
<td>229</td>
</tr>
<tr>
<td>9.10</td>
<td>Metric for measuring execution time of supervisors</td>
<td>229</td>
</tr>
<tr>
<td>9.11</td>
<td>Metric for computing percentage</td>
<td>230</td>
</tr>
<tr>
<td>9.12</td>
<td>Example of execution time SLO</td>
<td>230</td>
</tr>
<tr>
<td>9.13</td>
<td>Example of percentage SLO</td>
<td>230</td>
</tr>
</tbody>
</table>
## List of Figures

1. Conceptual system architecture ........................................... 6
2. Content-based Routing ..................................................... 22
2.2 PADRES broker architecture ........................................... 23
2.3 Compatibility graph ..................................................... 38
2.4 DAGs resulting from Figure 2.3 ........................................ 39
3. Client movement .......................................................... 46
3.2 P/S layers ................................................................. 47
3.3 Protocol overview ....................................................... 53
3.4 Client and coordinator states at source and target sites ........ 54
3.5 Global reachable states for local state graphs in Figure 3.4 .... 55
3.6 Default topology ........................................................ 61
3.7 Subscription covering relationships ................................... 61
3.8 Movement latency over time .......................................... 62
3.9 Subscription workload .................................................. 63
3.10 Number of clients ....................................................... 63
3.11 Single client ............................................................ 64
3.12 Incremental movement ................................................ 64
3.13 Topology size ............................................................ 65
3.14 Wide-area PlanetLab deployment ................................... 67
4. Example sub. propagation .................................................. 71
4.2 Relationships among $P(S)$, $P(NRS(S))$, $P(INC(S))$, and $P^*(S)$ 74
4.3 An example of a subscription relation graph .................... 77
4.4 An example of a simplified subscription relation graph ....... 79
4.5 Example publication propagation ..................................... 82
4.6 A subscription tree before and after pruning ...................... 85
4.7 Default broker topology ................................................. 86
4.8 Comparisons of active covering, lazy covering, subscription 90
packing, and incremental unsubscription algorithms ............ 86
4.9 Varying number of unsubscriptions .................................. 93
4.10 Varying subscription covering degree .............................. 94
4.11 Distributed business process scenario ............................. 97
4.12 Comparisons of active covering, lazy covering, packing and 99
4.13 Varying number of unsubscriptions (complex matching engine) . . . . . 99
4.14 Varying subscription covering degree (complex matching engine) . . . . . 99
4.15 Clustered workload (complex matching engine) . . . . . . . . . . . . . . 100
4.16 Tiered topology for the mobile scenario . . . . . . . . . . . . . . . . . . 101
4.17 Mobile scenario (complex matching engine) . . . . . . . . . . . . . . 104

5.1 Execution engine architectures . . . . . . . . . . . . . . . . . . . . . . 106
5.2 NIÑOS distributed business process execution architecture . . . . . . . 107
5.3 BPEL while activity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.4 BPEL pick activity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.5 BPEL compensate activity . . . . . . . . . . . . . . . . . . . . . . . . . 112
5.6 BPEL switch activity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
5.7 BPEL flow activity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
5.8 Process deployment and execution monitor . . . . . . . . . . . . . . . . . 118
5.9 Inter-organizational deployment . . . . . . . . . . . . . . . . . . . . . . 119
5.10 PADRES monitor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
5.11 Example loan application process . . . . . . . . . . . . . . . . . . . . . . 123
5.12 Varying request rates . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
5.13 Varying Web service execution time (50 requests/min) . . . . . . . . . 132
5.14 Varying Web service execution time (1000 requests/min) . . . . . . . . 133
5.15 Varying message size (50 requests/min) . . . . . . . . . . . . . . . . . . 134
5.16 Varying message size (500 requests/min) . . . . . . . . . . . . . . . . . . 134
5.17 Parallel business process . . . . . . . . . . . . . . . . . . . . . . . . . . 136
5.18 Average execution time . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

6.1 Distributed execution engine . . . . . . . . . . . . . . . . . . . . . . . . . 142
6.2 Activity profiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
6.3 Engine profiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
6.4 Metric estimator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
6.5 Redeployment manager . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
6.6 Cost model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150
6.7 Generated monitoring agents . . . . . . . . . . . . . . . . . . . . . . . . . 154
6.8 Experimental topology . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
6.9 Default workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156
6.10 Control flow hotspot . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
6.11 Varying control flow hotspot . . . . . . . . . . . . . . . . . . . . . . . . . 158
6.12 Varying candidate radius . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
6.13 Initial activity deployment for local optima experiments . . . . . . . . . 160
6.14 Activity set redeployment . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
6.15 Monitoring workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
6.16 Sensitivity optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
6.17 Larger workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
6.18 Deployment of workflow in Figure 6.17 . . . . . . . . . . . . . . . . . . . 163
6.19 Active engines with energy use cost function . . . . . . . . . . . . . . . . 164
7.1 Static resource discovery model ........................................ 168
7.2 Dynamic resource discovery model ................................... 170
7.3 Continuous resource discovery model ................................. 173
7.4 Subscription blocks ...................................................... 175
7.5 Similarity forwarding example ....................................... 179
7.6 Resource workloads ..................................................... 183
7.7 Discovery workload distributions ..................................... 184
7.8 Uniform spatial and uniform temporal discovery distribution .... 185
7.9 Zipf spatial and uniform temporal discovery distribution .......... 185
7.10 Uniform spatial and Gaussian temporal discovery distribution .... 185
7.11 Query time for centralized and decentralized architectures ...... 188
7.12 Decentralization message overhead ................................ 189
7.13 Direct and normal result delivery ................................... 189

8.1 System architecture ..................................................... 192
8.2 Example services and requests ....................................... 193
8.3 Dead search paths ....................................................... 200
8.4 Composition with loop ................................................ 200
8.5 Broker topology ....................................................... 202
8.6 Search result structures .............................................. 202
8.7 Search frequency ...................................................... 204
8.8 Results per search .................................................... 206
8.9 Process length ......................................................... 206
8.10 Service deployment ................................................... 208
8.11 Registered services .................................................. 209
8.12 Service similarity .................................................... 210

9.1 Overview of proposed solution ...................................... 213
9.2 Overview of SLA model .............................................. 213
9.3 Examples of scopes ................................................... 214
9.4 Credit check process .................................................. 217
9.5 SLA editor ............................................................ 222
9.6 Event propagation in SLA ............................................. 225
9.7 Overview of runtime architecture ................................... 226
9.8 Claim approval process .............................................. 227
9.9 Implementation of action handler ................................... 231
Chapter 1

Introduction

Applications are becoming increasingly distributed and loosely coupled in terms of their development processes, software architectures, and deployment platforms. For example, in Web mashups [101, 105], utility or cloud computing environments [6, 7], and service-oriented architectures (SOA) [21, 73] applications are developed by orchestrating reusable services using high level workflows or business processes.

One benefit of developing composite applications is the increased flexibility that arises from being able to develop applications by composing from an inventory of available services. As well, since service providers and consumers may be geographically dispersed, many composite applications are inherently distributed.

This thesis considers a number of phases in developing, executing, and monitoring these business processes and offers solutions to increase the flexibility of managing these processes and develops a distributed platform that better fits the distributed nature of the environment.

1.1 Motivation

Enterprise applications are increasingly being architected in a service-oriented architecture (SOA) style, in which modular components are composed to implement the business
logic [21, 73, 62, 91]. The properties of such applications, such as loose coupling among the modules, is promoted as a way for an agile business to quickly adapt its processes to an ever changing landscape of opportunities, priorities, partners, and competitors. The proliferation of Web services standards in this area reflects the industry interest and demand for distributed enterprise applications that communicate with software services provided by vendors, clients, and partners [26].

For example, an online retailer may utilize the services of a partner shipping company to allow their customers to track the delivery status of products. The shipping company here would expose a component that allows its partners to retrieve delivery status information. Other external services the retailer may use include a payment service (such as PayPal), or a product review database. In addition, the retailer may use services developed internally, such as a user interface engine (to render interfaces for various devices such as a PDA or PC), and an authentication service. Developing such loosely coupled components makes it easier to develop, maintain, and modify the application.

It is not uncommon for business processes in industries such as supply chain management, online retail, or health care to consist of complex interactions among a large set of geographically distributed services developed and maintained by various organizations. The processes themselves can be very large, long running, manipulate vast quantities of data, and require thousands or millions of concurrent process instances. For example, one of our project partners reports that a large Chinese electronics manufacturer employs formal business processes to drive its operations activities including component stocking, manufacturing, warehouse management, order management, and sales forecasting [57]. The processes are inherently distributed using department-level processes for manufacturing, warehouse and order management, with each of these processes utilizing from 26 to 47 activities. There also exist global processes that compose the department-level ones. In addition to the separation by administrative domains, the processes also involve geographically distributed parties including a number of suppliers, several organizational
departments, a dozen sales centers, and many retailers. Thousands of instances of these processes are executing concurrently at any one time.

Business processes are executed by an orchestration or execution engine that is responsible for carrying out the activities in the process, and maintaining the state associated with process instances. Typically a single engine is deployed to manage an application, and scalability is addressed by replicating the engine. Existing execution engines support clustering in order to optimize and ensure business process throughput on highly available systems [39, 76]. When a business process needs to be scaled to meet heavier processing needs, the engine’s clustering algorithm automatically distributes processing across multiple engines. This thesis, on the other hand, develops a distributed process execution architecture, one that is more in agreement with the distributed nature of the processes themselves.

The distributed executing engine developed here, called NIÑOS, orchestrates business processes by distributing process execution across several light-weight agents, each of which carry out a single activity in the process. This distributed architecture is congruent with an inherently distributed enterprise where business processes are geographically dispersed and coordinating partners have to communicate across administrative domains. Not only does the design remove the scalability bottleneck of a centralized execution engine, 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. Furthermore, NIÑOS supports flexible mappings of the process activities onto heterogeneous platforms and resources, permitting the system to shape itself from a centralized to a fully distributed configuration. The distributed design also allows portions of a process to be delegated to a partner, a common task for businesses that outsource parts of their operations.

While the NIÑOS architecture provides a distributed execution platform, and supports flexible process deployment opportunities, it still requires manual effort to decide
Chapter 1. Introduction

where activities should be deployed. In some cases, this decision may be obvious based on security or administrative constraints, but an administrator may also have the flexibility to place activities to achieve certain non-functional goals, such as minimizing network traffic or energy use. These goals are often expressed as Service Level Agreements (SLA) [38, 78] which define a contract between a service provider and consumer. This thesis develops an SLA-driven approach to dynamically determine the placement of activities across the available execution engines in order to satisfy the SLA.

In order to facilitate a dynamic process execution platform where processes are frequently modified or redeployed, this work utilizes a distributed content-based publish/subscribe overlay. While this publish/subscribe layer offers flexible, and decoupled interactions among the clients in the system, it is extended with support for an atomic movement operation. This allows the system to redeploy activities without interrupting the process. As well, since distributed publish/subscribe protocols may be susceptible to congestion triggered by a form of client churn, this thesis develops a distributed algorithm to detect and avoid such congestion.

Another important concern when developing distributed composite applications is to find a required service from an catalogue of existing services. For example, when authoring a business process to handle an online product order, the developer may wish to use a bank’s payment service to carry out a credit card transaction. In popular domains the available services may be annotated with structured metadata so developers can query for services that match their criteria. For example, among the payment services available, a developer may want to find one that is cheap, supports certain credit cards, and offers some reliability guarantees. Moreover a rich service description may make it possible to dynamically find services at runtime and substitute the best service available while the process is executing. While many existing techniques exist to register and find services [82, 27, 86, 37], this thesis develops a distributed event-based algorithm that can not only query a distributed set of service registries, but also continuously monitor these
registries and report when a new matching service is found.

Sometimes a developer may not find an individual service that performs a desired action, and is then forced to create a composition of services. This can be a labour intensive task, only made worse with wider adoption of service computing and proliferation of the number of available services that may be composed. For example, a user wishing to print an HTML page, may search for a process that accepts as input an HTML document, and returns as output the status of the print job. However, it may be that the printer service, \( W_p \), only accepts a PostScript file as input, but there is another service, \( W_c \), that can convert an HTML document into a PostScript file. In this case, the search result would be a process consisting of the two services \( W_p \) and \( W_c \) executed in series. A more complex example is a property developer who wants to obtain a permit to build a condominium in a city. While this is a common process, the developer may not be familiar with the city’s laws, and does not want to manually examine the many Web services offered by the municipality. Instead, the developer searches for a composition of services whose input is a request for a permit, and whose output is a building permit. A process may be found that composes services to first obtain zoning clearances, followed by an environmental assessment, and only then request electricity, water, and sewage approvals (perhaps in parallel). Finally, a service collects the approvals from the various utilities, and returns a building permit to the user.

What is needed is a way for the user to search for processes, or service compositions, that meet some functional criteria, allowing the user to then examine the results and select appropriate processes for direct use or further tuning. This problem is known as \emph{automatic service composition}, and has been an active field of research \cite{10, 11, 46, 49, 59, 72, 75, 100, 108} and has garnered interest in industry \cite{60, 84}. Existing automatic service composition algorithms are centralized \cite{63}, and may have difficulty scaling to large service repository sizes, many concurrent searches, or distributed repositories. Therefore, this thesis develops a distributed service composition algorithm
Figure 1.1: Conceptual system architecture

that offers a more appropriate architecture for service computing systems that are already distributed, with services scattered geographically, administered by disparate administrative domains, and registered with several autonomous registries. Moreover, distributed search has the potential to address certain shortcomings of centralized algorithms, including scalability, parallelism, and the ability to more efficiently utilize the resources available in a distributed environment.

In an environment where processes are composed of services purchased from different parties, it becomes important to ensure that these services provide the expected SLAs agreed upon by both the service provider and consumer [42]. To ensure no party violates an SLA, the business process must be constantly monitored to report any SLA violations. Implementing an automated monitoring solution by assembling various technologies can require considerable effort and expertise, and the resulting solution is non-standard, often ad-hoc, error-prone, and not necessarily optimized for performance. This thesis simplifies SLA authoring and monitoring with a flexible SLA model and distributed monitoring architecture. The SLA model is loosely coupled with the referencing business process
allowing each to be evolved independently. As well, monitoring artifacts can be automatically generated from the SLA model, and deployed over a distributed architecture to detect violations of the SLA at runtime.

The architecture of the system developed in this thesis is conceptually summarized in Figure 1.1. A set of computing and network resources are virtualized by a distributed content-based publish/subscribe routing layer. Over this layer is a distributed business process execution engine, and in the top layer are various tools to monitor the underlying processes, or find and compose services.

1.2 Problem statement

This thesis focuses on developing more distributed and flexible solutions for the management of business processes. To this end, it addresses three sets of problems addressing each layer in the system architecture depicted in Figure 1.1. First, it extends the distributed content-based publish/subscribe overlay with support for more dynamic client interactions. Next, it builds a flexible and distributed business process execution engine on top of this publish/subscribe overlay. Finally, it provides facilities to manage and monitor services, specifically distributed algorithms to discover and compose services in the system, and the ability to specify and monitor SLAs. Each of these three sets of problems are further detailed below.

1.2.1 Dynamic publish/subscribe client interactions

In an agile environment, business processes may need to be redeployed or modified frequently. The redeployment of process activities requires a new primitive to move publish/subscribe clients according to certain transactional properties. Similarly, the modifications to a process or changes in how the process is monitored require efficient processing of unsubscription messages that may otherwise destabilize the system.
Chapter 1. Introduction

Transactional client movement

Many adaptive distributed applications require the stateful reprovisioning of software components. For example, applications on virtual grid infrastructures are redeployed based on application requirements and grid conditions [30], massive multiplayer games migrate game state among servers in response to changing workloads [12], distributed process execution systems dynamically schedule agents at various engines [71], adaptive stream processing engines reconfigure dataflow operators to optimize query execution [47, 83], load balancing algorithms move software modules between nodes [107], mobile agent frameworks migrate program code and state across nodes [66, 31], and in mobile applications such as location-based services, the nodes themselves are mobile and connect to nearby access points as they roam.

Distributed reprovisioning of publish/subscribe applications requires the ability to move publish/subscribe clients in a well-behaved manner. For example, moving clients should neither miss messages nor receive duplicates, and should be able to send messages without interruption. To achieve this transparency, a protocol must provide guarantees at the publish/subscribe layer during routing table reconfiguration. However, there is little literature that develops well-defined transactional movement properties in a distributed content-based publish/subscribe network. Though some work on publish/subscribe mobility has been put forward [25, 70], they have used end-to-end protocols that relied on optimizations, such as subscription covering, to achieve efficiency. As we demonstrate in this thesis, even with these optimizations, such protocols can be too costly in an environment characterized by frequent movement. We analyze the problems and shortcomings of traditional movement protocols, and propose an alternative protocol.

Incremental filter aggregation

Distributed publish/subscribe systems can be vulnerable to congestion triggered by churn in the client interests, even when clients are stationary. A change in a client’s interest is
expressed as a subscription update in a publish/subscribe system, and naturally occurs in many application scenarios. For example, changes to a business process will alter the communication patterns among the corresponding activities and require updating their interests. Similarly, it is common that different aspects of a system need to be monitored at different times, and each change in the monitoring requirements will also require the publish/subscribe clients’ interests to be modified.

The congestion induced by this type of churn is a consequence of the common subscription covering technique in which a general subscription aggregates the interests of the more specific ones it covers [65, 16]. The covering optimization has many benefits including smaller routing tables, fewer subscription propagations, and faster routing decisions [52, 16]. What is not well known is that these techniques have a drawback that manifests itself in an environment with dynamic client interests. Consider a content-based router with a general subscription $S$ and a set of more specific subscriptions $\mathcal{S}$ that are covered by $S$. The covering optimization will only forward $S$ and avoid forwarding the subscriptions in $\mathcal{S}$. However, when $S$ is to be removed, perhaps because the client is not interested in $S$ any more, the subscriptions in $\mathcal{S}$ are no longer covered by any subscription, and it becomes necessary to forward all the subscriptions in $\mathcal{S}$. To maintain correctness of the routing tables, all the subscriptions in $\mathcal{S}$ are forwarded immediately before removing $S$. As it is costly to record the covering relationships among subscriptions [94], a large set $\mathcal{S}$ can impose severe congestion in the network, overwhelming the brokers that must process this burst of subscription traffic, and destabilizing the system.

Consider a distributed business process management system, where each process activity would issue subscriptions that reflect its narrow interest in the events that trigger it [57, 103, 67]. As well, an administrator wanting to monitor the entire system would issue a broad subscription $S$ that covers many of the existing ones. When monitoring stops, the removal of the associated subscription $S$ will result in a large and instantaneous burst of many subscriptions that were heretofore covered by $S$. 
Chapter 1. Introduction

The goal is to develop mechanisms to avoid congestion-induced instability of the content-based routing protocols while preserving the benefits of subscription aggregation techniques. There is an inherent tradeoff here. For example, in the above scenario, a trivial solution to avoid the burst of covered subscriptions in $S$ is to retain the covering subscription $S$ even after a client has indicated it is not interested in $S$. However, this results in a situation where a message that satisfies $S$ but not any in $S$ will be unnecessarily propagated. Depending on the message rates and the similarity of interest between $S$ and $S$, this false positive traffic can impose significant unnecessary network and processing load on the system. At the other extreme are existing routing protocols [8], which immediately replace $S$ with $S$, thereby avoiding false positive message traffic but leaving the system susceptible to bursts of subscription propagations.

The solution in this thesis navigates the solution space that lies between these two extreme possibilities. In particular, the solution selectively preserves or replaces portions of the subscription propagation tree with the more precise, but also more numerous, covered subscriptions. When a client removes a subscription $S$, a distributed algorithm is used in which each router determines whether it is better to preserve or remove $S$. The decision to preserve $S$ depends on the similarity between $S$ and the subscriptions that would replace it. If the similarity is high, there will be few wasted false positive messages, and it is worthwhile to preserve $S$.

1.2.2 Distributed process execution

Transitioning now to the second layer in Figure 1.1, this thesis develops a distributed execution engine called NIÑOS to support the inherently distributed nature of a service-oriented architecture. As well, a distributed algorithm is devised to optimize the placement of activities among the distributed set of engines.
Distributed execution engine

Unlike traditional centralized or clustered execution engines, a distributed engine would decompose a process into individual activities and allow the deployment of these activities to be distributed. Such an architecture can utilize additional resources for better scalability, and allows for more flexible deployment strategies. For example, it becomes possible to strategically co-locate certain activities with the data it accesses. As we will see, our distributed architecture also naturally enables the fine-grained monitoring of a process without additional instrumentation. Moreover, a distributed engine is desirable when processes are inherently distributed across administrative domains and geographic locations [57]. In these cases, a distributed process execution engine allows flexibility in deploying portions of a process to the associated departments, or to outsource parts of a process to other organizations.

NIÑOS utilizes and exploits the rich capabilities of the PADRES distributed content-based publish/subscribe routing overlay [29, 54, 56, 36, 41]. All communication in the system occurs as publish/subscribe interactions, including process coordination, control and monitoring. This decouples the NIÑOS components, which now only need to be aware of one another’s content-based addresses, thereby simplifying agent reconfiguration and movement, and seamlessly allowing multiple processes and process instances to coexist. In addition, in NIÑOS, processes are transformed such that certain computations are carried out in the publish/subscribe layer, exploiting advanced features available in PADRES. This further simplifies the orchestration, and allows these computations to be optimized by the PADRES layer by, for example, performing in-network event correlation. Yet another advantage afforded by the publish/subscribe layer is ease of administration. Components can be configured and controlled individually, or as some subset, using their location-independent content-based addresses. Similarly, since all communication occurs over the publish/subscribe layer, the system can be fully monitored without additional instrumentation logic. The declarative publish/subscribe interface supports expressive
queries for precisely the information of interest.

**Distributed activity placement**

Business process management (BPM) practices address complexity in enterprise environments with systematic development processes [98]. However, the development, administration and maintenance of a business process still requires much manual effort that can be automated. For example, non-functional goals, often expressed as Service Level Agreements (SLA) [38, 78] defining a contract between a service provider and consumer, are specified during an early design stage but may need to be manually considered at each subsequent stage of development. The requirement that process instances complete within some specified time, for instance, may influence decisions in the development, deployment, resource provisioning, and monitoring of the process.

We present an SLA-driven approach to BPM for service-oriented applications in environments such as cloud computing platforms. The approach employs two key ideas: a cost model, and a distributed event processing platform. The cost model includes an unambiguous cost function—derived from a user’s SLA—that the system uses to make process deployment and monitoring decisions consistent with the SLA. The event processing platform is used to develop a scalable event-based distributed process execution engine that supports flexible and adaptive process deployment options. These ideas complement one another in that the goals specified in the SLA are used to automatically monitor the relevant parts of a process’s execution in a loosely-coupled manner, and optimize the deployment of the process at runtime.

### 1.2.3 Service management and monitoring

Considering the management layer in Figure 1.1 now, an important concern in developing distributed applications is to find services that match certain functional and non-functional requirements, and then compose these services into a useful process. This
thesis develops distributed algorithms to search and compose services that are registered in a distributed set of service repositories.

Service and resource discovery

Many methods for service discovery have already been proposed [86, 92, 37]. However, these approaches focus on the routing mechanism of discovery and have not paid much attention to the inherent static and dynamic characteristics of service discovery. Services or resources always have static attributes (like the CPU and memory configuration of a server), but also have dynamic attributes that change frequently (such as the available disk storage). Current approaches lack flexibility, thus limiting their use, as they do not consider the difference in static and dynamic attributes characterizing resources’ capabilities.

In this thesis, an event-based resource discovery approach is proposed that combines static and dynamic resource discovery, and adds a third continuous discovery model. The approach exploits a distributed content-based publish/subscribe system to achieve scalable, efficient, and real-time resource discovery. Resource registrations, discovery requests, and results are all mapped to publish/subscribe messages. The “push” capabilities of the publish/subscribe model is also what allows for a powerful continuous discovery model where users can be notified in real-time of new resources that match their criteria.

In practice, it is expected that there is a degree of similarity among resource requests, similar to the findings on Web requests following a Zipf distribution [14], the discovery approach in this thesis is optimized by sharing results among concurrent requests with similar interests. Again, publish/subscribe techniques, namely subscription covering, are used to help find similar requests and share their results.
**Automatic service composition**

Automatic service composition algorithms search a repository of available services and return compositions of services that meet some user defined criteria. Often the criteria indicates some input and output requirements. Many algorithms have been proposed to solve the problem of automatic service composition using techniques such as AI planning and theorem proving [63].

As existing centralized service composition algorithms may not be appropriate for a distributed service-oriented environment, this thesis develops a distributed automatic service composition algorithm. The algorithm must be able to search and compose services registered across a distributed set of service registries, and distribute processing overhead among these registries.

This thesis uses a publish/subscribe system [16, 25, 29] to solve the process search problem. It turns out that determining the interaction relationships among the data senders and receivers in a publish/subscribe system bears some similarity to discovering possible service compositions. Therefore, with an appropriate mapping of the problem space, the data structures and algorithms used by a publish/subscribe broker for publish/subscribe matching can be exploited for service composition. This is a unique use of a publish/subscribe system and allows us to take advantage of existing work on efficient and scalable publish/subscribe algorithms.

**SLA modeling and monitoring**

To monitor an SLA, the developer must first interpret the SLA, then instrument the associated process so that the required metrics can be measured, and finally implement the logic to gather and process these measurements and report SLA violations. In existing approaches, the SLA may be tightly coupled with the process, may be time consuming to specify, or even require the process to be substantially modified to monitor the required SLA [44, 50, 88]. This is usually not desirable because it discourages the reuse of the
SLA and is difficult to extend and maintain both the SLA and the process.

This thesis develops a flexible model that allows SLAs to be specified in a manner that is loosely coupled with its associated process. The SLA model is modular and allows an SLA to be easily specified by composing a number of SLA components. As the relationship between the SLA and process may be broken during the parallel development of the two artifacts, the SLA can be validated at design time to identify the broken dependencies. It is assumed that the process engine emits events as the process is executed, as is the case with the NIÑOS system presented in this thesis. This design allows for the automatic generation of code to consume the relevant events and monitor for SLA violations.

1.3 Contributions

This thesis makes a total of seven major contributions: two contributions for each of bottom two layers in Figure 1.1, and three contributions for the top management layer. First are the extensions to the content-based publish/subscribe overlay to support the efficient movement of clients [36, 70] and changes in client interests [64]. Next are the development of a distributed process execution engine [57, 55] and distributed activity placement algorithms [69]. The final set of contributions address searching [106] and composing [34] services in a distributed manner, and modeling and monitoring SLAs [22]. Each contribution is outlined in more detail below.

1. The first contribution introduces a transactional movement primitive to a distributed content-based publish/subscribe system [36]. A set of formal properties are defined for this primitive, and efficient protocols are implemented to support these properties. Evaluations of a real implementation of the protocols show that they can complete a movement faster than traditional protocols, require less network overhead, and have more predictable behaviour under a variety of scenarios.
2. The next contribution also concerns the publish/subscribe layer, and addresses congestion that may arise due to changes in subscriptions from publish/subscribe clients [64]. A distributed protocol compares the congestion triggered by the removal of a subscription with the false positive traffic of preserving the subscription, and incrementally prunes the propagation tree of the subscription. Results consistently show that the protocol provides significant benefits by avoiding severe network congestion that otherwise persist for several hours, reducing routing table sizes by about 50%, and decreasing message propagation delays by several orders of magnitude.

3. Building on the publish/subscribe layer extensions above, the next contribution develops a distributed process execution engine called NIÑOS [57]. The execution engine maps processes authored in the standard Business Process Execution Language (BPEL), including the complete set of BPEL activities, to a set of distributed NIÑOS agents, with control flow realized using decoupled publish/subscribe semantics. The distributed architecture supports more flexible process deployment and management options. As well, evaluations of the NIÑOS system demonstrates its improved scalability over a centralized engine.

4. The next contribution develops protocols to automatically redeploy processes in the NIÑOS architecture at runtime to satisfy a user-specified Service Level Agreement (SLA) [69]. Each execution engine updates a cost model using profiles gathered from a neighbourhood of activities and engines in the system, and makes independent decisions on where to relocate certain activities. The transactional movement primitive introduced earlier is used to move activities without interrupting the process. In addition, a case study addresses the problem of process monitoring: by systematically mapping the monitoring requirements of a process to a process itself, the above algorithms can be reapplied to optimize the deployment of the agents.
monitoring a process. Evaluations demonstrate these algorithms can achieve significant savings.

5. Moving now to the management of processes in the system, this thesis develops a unified static and dynamic service discovery framework based on a content-based publish/subscribe overlay [106]. A new continuous discovery model is developed that allows for efficient real-time notifications of newly registered matching services. As well, an algorithm is devised to optimize discovery performance by sharing results among similar discovery requests. A detailed evaluation under variety of workloads shows that the system can effectively exploit similarities in search queries to reduce discovery time and overhead.

6. A related contribution develops a distributed solution to the problem of automatic service composition [34]. Service interface descriptions are modeled as publish/subscribe messages in such a way that the matching of publish/subscribe messages also indicates service composition relationships. A distributed process search algorithm uses the relationships discovered above to incrementally compose services into a process that satisfies the user’s process interface criteria. Evaluations reveal that the algorithm can outperform a centralized one under various conditions.

7. The final contribution develops a flexible model to author SLAs that specify the quality of service guarantees negotiated by service providers and consumers. [22]. The model extends the WSLA language and attempts to minimize the dependencies between the SLA and associated process, as well as check for broken dependencies. The model encodes sufficient information to automatically generate artifacts to monitor the SLA, and these monitoring artifacts are deployed across a distributed publish/subscribe platform. The design supports distributed monitoring, reuses monitoring logic across SLAs with common subexpressions, and does not require additional instrumentation of the process even when the SLA is modified.
1.4 Organization

The remainder of this thesis is organized as follows. First, Chapter 2 presents some background and related work relevant to the contributions in this thesis. This includes a discussion of the publish/subscribe literature and a description of the PADRES publish/subscribe system used in this work. Next the three sets of problems outlined above are addressed across six chapters, with two chapters devoted to each problem set. First are the extensions to support dynamic client interactions: Chapter 3 develops properties and algorithms to move clients according to certain transactional properties, and Chapter 4 addresses the efficient management of changing client interests. Next, is the development of a distributed process execution engine: Chapter 5 presents a distributed orchestration engine using publish/subscribe primitives, and Chapter 6 describes algorithms to place activities in order to satisfy user-specified goals. Finally, the problems of managing services in a distributed service registry and monitoring processes are addressed: Chapter 7 defines a number of service discovery models and implements corresponding algorithms for each model, Chapter 8 designs protocols to automatically compose a set of services into a process that conforms to a user-specify interface description, and Chapter 9 develops a loosely-coupled SLA model and distributed process monitoring architecture. The thesis concludes with Chapter 10 presenting a summary of the key points of this work and a discussion of future research directions.
Chapter 2

Background and related work

This chapter presents an overview of the ideas that this thesis builds on, and positions the work here with respect to the literature.

2.1 Content-based publish/subscribe

2.1.1 Matching semantics

The algorithms in this thesis build on a publish/subscribe messaging substrate which provides a flexible and powerful interaction model for a wide variety of large-scale distributed systems. The publish/subscribe model consists of three basic elements: subscribers, who express interest in particular information by means of a subscription language; publishers, who publish information of interest; and a broker or broker network which is responsible for matching publications with subscriptions and for routing publications to interested subscribers. The brokers in the model decouple the publishers and subscribers in space and time making the publish/subscribe paradigm particularly well-suited for large and dynamic distributed systems. One dimension along which to compare publish/subscribe systems is the expressiveness of their subscription language. In channel-based publish/subscribe, publications are sent to a predefined channel, and subscribers interested in a
Table 2.1: Examples of subscriptions, advertisements and intersection relations

<table>
<thead>
<tr>
<th>Subscription $s$</th>
<th>Advertisement $a$</th>
<th>Intersection relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\text{product} = \text{“computer”}, \text{brand} = \text{“IBM”}, \text{price} \leq 1600)$</td>
<td>$(\text{product} = \text{“computer”}, \text{brand} = \text{“IBM”}, \text{price} \leq 1500)$</td>
<td>$a$ intersects $s$</td>
</tr>
<tr>
<td>$(\text{product} = \text{“computer”}, \text{price} \leq 1600)$</td>
<td>$(\text{product} = \text{“computer”}, \text{brand} = \text{“IBM”}, \text{price} \leq 1600)$</td>
<td>$a$ intersects $s$</td>
</tr>
<tr>
<td>$(\text{product} = \text{“computer”}, \text{brand} = \text{“IBM”}, \text{price} \leq 1600)$</td>
<td>$(\text{product} = \text{“computer”}, \text{brand} = \text{“Dell”}, \text{price} \leq 1500)$</td>
<td>$a$ does not intersect $s$</td>
</tr>
</tbody>
</table>

Some systems use advertisements to let publishers announce the set of publications they are going to publish. An advertisement $a$ matches a publication $e$ if and only if all attribute-value pairs match some predicates in the advertisement. Formally, an advertisement $a = \{p_1, p_2, \ldots, p_n\}$ determines a publication $e$, if and only if $\forall (\text{attr}, \text{val}) \in e$, there exists a predicate $p_k \in a$ where $(\text{attr}, \text{val})$ matches $p_k$.

An advertisement $a$ intersects a subscription/advertisement $s$ if and only if the intersection of the set of the publications determined by the advertisement $a$ and the set of the publications that match $s$ is a non-empty set. Formally, at the predicate level, an advertisement $a = \{a_1, a_2, \ldots, a_n\}$ intersects a subscription/advertisement $s = \{s_1, s_2, \ldots, s_n\}$
if and only if $\forall s_k \in s, \exists a_j \in a$ and there exists some attribute-value pair $(attr, val)^1$ such that $(attr, val)$ matches both $s_k$ and $a_j$. Table 2.1 presents some examples of subscriptions and advertisements and the corresponding intersection relations.

2.1.2 Content-based routing

In an advertisement-based publish/subscribe system [29, 16], an advertisement specifies the information that the publisher may publish in the future. Advertisement messages, flooded throughout the network, build a spanning tree rooted at the publisher, and serve to direct the routing of subscriptions only towards publishers whose advertisements are of potential interest to subscribers’ subscriptions [53]. Finally, publications are delivered to interested subscribers along the paths built by subscription messages.

Figure 2.1 shows an example of content-based routing with advertisements in the PADRES publish/subscribe system. The subscription routing table (SRT), consisting of $\langle$advertisement, lasthop$\rangle$-tuples, is used to route subscriptions, and likewise, the publication routing table (PRT) stores $\langle$subscription, lasthop$\rangle$ tuples that are used to route publications towards interested subscribers. For example, in Figure 2.1, advertisement $adv_1$ is broadcast throughout the network and stored at each broker with the appropriate lasthop. Subscriptions that match $adv_1$ are routed according to these lasthops; for example, $sub_1$ is routed along the Path $B - C - A$. Note that the subscription $sub_1$ is not forwarded to Broker $D$ since $adv_1$ indicates that matched publications are from Broker $A$. Therefore, publication $pub_1$ is routed along the reverse Path $A - C - B$ to the subscriber.
2.1.3 PADRES broker

The PADRES brokers are modular software components built on a set of queues: one input queue and multiple output queues, with each output queue representing a unique message destination. A diagram of the broker internals is provided in Figure 2.2. The matching engine, a critical component of a broker, maintains various data structures. In one such structure subscriptions are mapped to rules, and publications are mapped to facts. The rule engine performs matching and decides the next-hop destinations of the messages. This novel rule-based routing approach allows for powerful subscription semantics and naturally enables composites subscriptions, which are more complex rules in the rule engine. Mapping the subscription language to a rule language is relatively straightforward, and extending this subscription language does not require significant changes in the engine. Furthermore, rule engines are well-studied, allowing PADRES to take advantage of existing research. Our experience with the system indicates that

\[ s_k \text{ and } a_j \text{ refer to the same attribute } \text{attr}. \]
2.1.4 Covering and merging

Aggregating subscriptions and advertisements in a content-based network using a covering optimization has been implemented in many publish/subscribe systems [16, 29]. Given two filters (whether advertisements or subscriptions), a filter $F$ covers a filter $G$ denoted by $F \supseteq G$, iff $N(F) \supseteq N(G)$, where $N(X)$ is the set of publications that match $X$. Based on this definition, similarities among subscriptions can be found and used to remove redundant subscriptions from the network, maintain compact routing tables and reduce network traffic congestion.

Consider the example in Figure 2.1 when a new subscription $sub_3$ is issued from a subscriber connected to Broker $D$. If subscription $sub_3$ matches advertisement $adv_1$, $sub_3$ will be routed along the Path $D-C-A$. However, if subscription $sub_1$ covers subscription
sub₃ (such as sub₃ ([class,eq,abc],[a,¿,18])) at Broker C, sub₃ is not forwarded to Broker A. All publications matching sub₃ must also match sub₁. A formal definition of the covering relation is as follows: A subscription sub₁ covers sub₃, if and only if, \( P(sub₁) \supseteq P(sub₃) \) (where \( P(s) \) refers to the set of publications that match subscription \( s \)), denoted as \( sub₁ \supseteq sub₃ \). The covering relation defines a partial order on the set of all subscriptions with respect to \( \supseteq \). The covering relations among advertisements can be defined in a similar manner.

In the PADRES system used in this thesis, two variants of the covering optimization are implemented. The first one is called active covering, and strictly obeys the classic covering definition from [16, 65]. With active covering, when a subscription \( S' \) arrives at a broker after a subscription \( S \) that covers \( S' \), the broker does not forward \( S' \). Moreover, if \( S' \) arrives before \( S \), \( S \) is forwarded, and also brokers that see both \( S' \) and \( S \) delete \( S' \) from their routing tables. This helps to ensure more compact routing tables, but requires more processing and network traffic to clean up unnecessary \( S' \) routing state. Under lazy covering on the other hand, in the case where \( S' \) arrives before \( S \), \( S \) is simply forwarded and none of the \( S' \) routing state is cleaned up. This is a cheaper operation but results in larger routing tables over time.

Although covering-based routing is good at aggregating subscriptions and maintaining a compact routing table and reducing network traffic, the experiments in our earlier work [36] show that the more the subscriptions are covered, the worse the system will behave when certain subscriptions are removed. This problem is especially severe for active covering, but is also present with lazy covering.

The merging technique is used to further reduce the routing table size and the traffic overhead in the content-based network, and is used in addition to the covering optimization [16].

Formally, a filter \( F \) is a merger of a set of filters \( F₁, ..., Fₙ \), iff \( N(F) \supseteq \bigcup_{i=1}^{n} N(Fᵢ) \). There are two kinds of mergers. When the publication set of the merger is exactly equal
to the union of the publication sets of the original filters, \( N(F) = \left( \bigcup_{i=1}^{n} N(F_i) \right) \), it is a perfect merger. Otherwise, the publication set of the merger is larger than the union, it is an imperfect merger. Imperfect merging can reduce the number of subscriptions, but may allow publications to be forwarded that do not match any of the original subscriptions. In order to apply merging, it must be possible to efficiently compute mergers and if imperfect merging is performed the number of the unwanted publications must be small. While Crespo et al. [24] proposed merging of queries that are evaluated periodically against a database, they showed that in the general case query merging is NP-hard.

### 2.1.5 Client mobility

Publish/Subscribe research has focused on developing efficient routing protocols [8, 16], fast matching algorithms [28], and features such as failure handling [41], load balancing [107], and client mobility [25, 70]. However, none address the problem of supporting transactional client movement guarantees in the publish/subscribe system. The work presented in this thesis is therefore orthogonal to these approaches and presents an important and fundamental addition to the body of publish/subscribe knowledge. This thesis also establishes the important and surprising observation that the popular publish/subscribe covering optimizations may actually negatively affect performance.

Transaction models for publish/subscribe that define a transactional context for the processing of publications at subscribers have been proposed to simplify the development of event-based applications [61, 99]. The approaches are based on a central transaction coordinator, much like traditional transaction processing systems, and little experimental evidence to support the scalability of the approaches are presented. Our work addresses transactions in the context of mobile publish/subscribe clients aiming to preserve certain properties while clients move, which none of the prior approaches address.

Protocols to repair failures in publish/subscribe networks do not address routing table reconfigurations due to voluntary client movement [41, 79]. In the latter case
broker state changes result from announced movement, not unexpected failures, and
more efficient algorithms can be developed.

Mobility in publish/subscribe has mostly dealt with subscriber mobility, where the
broker network stores and replays publications missed by a moving subscriber [25, 15,
70]. In prior work we define and evaluate end-to-end subscriber and publisher mobility
protocols that rely on advertisement covering for efficiency [70]. A moving publisher
issues an advertisement (unadvertisement) at the new (old) access point. By contrast,
the work presented in this thesis performs reconfigurations in a more efficient hop-by-
hop manner that is also less susceptible to interactions among clients than an end-to-end
approach.

2.1.6 Congestion control

Congestion control in pub/sub systems differs from traditional congestion control found
in other networking systems. Pietzuch et al. [80] presented a congestion control scheme
based on the publish/subscribe messaging model, which was a combination of two conges-
tion control mechanisms, the subscriber host broker-driven protocol and publisher host
broker-driven protocol. This technique can effectively adjust the rate of publishing new
messages, allowing brokers under recovery to eventually catch up, and other brokers to
keep up. Although this thesis successfully handled the problem of congestion caused by
publications, it does not address the problem of congestion triggered by unsubscriptions.

2.2 Business processes and workflows

2.2.1 SOA development cycle

The service-oriented architecture (SOA) development cycle consists of modeling, develop-
ment, execution, and monitoring stages. Each stage differs in the level of abstraction and
is performed by the indicated roles, each of whom have varying expertise and concerns.
In the modeling stage of the development cycle, the business analyst would define the above process, abstracting from technology, infrastructure, and implementation details. The result of the modeling stage is an abstract model of the process represented in BPEL, BPMN, or some other process definition language. This representation is imported into the development stage, where architects and developers break the model into development artifacts such as services along with their interfaces, and implement the required business logic. The result of this stage is a set of deployable components that are represented in a standard specification such as the Service Component Architecture (SCA) model [77]. These services are deployed in a runtime environment that is managed by an administrator who is responsible for ensuring resource allocations and physical resource provisioning sufficient for the goals of the deployed services and processes. Often it is desirable to monitor the execution of the processes by tracking metrics on the state of the executing system. These metrics are computed based on observations in the runtime system and can be captured using standard monitoring frameworks such as an implementation of the Web Services Distributed Management standard [74]. The metrics gathered can be aggregated and presented to the stakeholders in the preceding development stages. For example, the business analyst may be interested in high level metrics such as the number of times the second credit check is required. On the other hand, the system architect may be interested in lower level metrics such as the processing delays of the individual credit checking services, while the administrator would be concerned with system performance bottlenecks such as network congestion or processor utilization.

In the modeling stage, high level, declarative goals are specified by the analyst, such as the throughput requirements of the process, or the cost constraints on the process. These goals are formalized into Service Level Agreements that specify a contract between the service provider and consumer. SLAs can be represented at different levels of abstractions, and may simply be a document at the modeling stage. These SLA documents are passed down the chain of the development process, with each stakeholder being re-
Chapter 2. Background and related work

sponsible for ensuring conformance to the SLAs. The architects and developers interpret and ensure that the services developed conform to the SLAs. During execution, SLA conformance is often achieved by over-provisioning resources and manually tuning the system. Finally, monitoring subsystems are instantiated to verify that the SLA goals are met, and that violations are reported to the appropriate parties. These violations are manually addressed by changes to the process, redevelopment of services, or provisioning of resources.

2.2.2 Business Process Execution Language

The Business Process Execution Language (BPEL) standard supports writing distributed applications by composing, or orchestrating, Web services. A BPEL process consists of a set of predefined activities. BPEL programs have properties of traditional programming languages (with concepts of scope, variables, and loops) and workflows (with concepts of parallel and sequential flows). BPEL processes are often authored in a proprietary graphical tool that serializes the process into a standard BPEL XML file.

BPEL activities can be classified as basic activities that perform some primitive operation such as receiving a message or throwing an exception, and structured activities that define control flow. The key BPEL activities are summarized in Table 2.2.

Several vendors have implemented BPEL engines, including IBM, Microsoft, Oracle, and Sun Microsystems. Scalability is typically addressed by load balancing process instances across a cluster of engines, where each engine still executes the entire process. In NIÑOS, however, the individual activities within a process are distributed among the available computing resources. The latter design also allows placing computational activities near the data they operate on, which is not possible in the cluster architecture. Furthermore, NIÑOS is applicable to the realization of cross-enterprise business process management, where no one single entity runs and controls the entire business process, but rather the process emerges as a choreographed concert of activities and sub-processes.
Basic Activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive</td>
<td>Blocking wait for a message to arrive.</td>
</tr>
<tr>
<td>reply</td>
<td>Respond to a synchronous operation.</td>
</tr>
<tr>
<td>assign</td>
<td>Manipulate state variables.</td>
</tr>
<tr>
<td>invoke</td>
<td>Synchronous or asynch. Web service call.</td>
</tr>
<tr>
<td>wait</td>
<td>Delay execution for a duration or deadline.</td>
</tr>
<tr>
<td>throw</td>
<td>Indicate a fault or exception.</td>
</tr>
<tr>
<td>compensate</td>
<td>Handle a fault or exception.</td>
</tr>
<tr>
<td>terminate</td>
<td>Terminate a process instance.</td>
</tr>
</tbody>
</table>

Structured Activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>Sequential execution of a set of activities.</td>
</tr>
<tr>
<td>while</td>
<td>Looping constructs.</td>
</tr>
<tr>
<td>switch</td>
<td>Conditional exec. based on instance state.</td>
</tr>
<tr>
<td>pick</td>
<td>Conditional exec. based on events.</td>
</tr>
<tr>
<td>flow</td>
<td>Concurrent execution.</td>
</tr>
</tbody>
</table>

Table 2.2: Summary of BPEL activities

run by each organization.

2.2.3 Distributed workflows

Distributed workflow processing has been studied in the 1990s to also address scalability, fault resilience, and enterprise-wide workflow management [5, 103, 67]. Alonso et al. present a detailed design of a distributed workflow management system [5]. The work bares similarity with our approach in that a business process is fully distributed among a set of nodes. However, the distribution architectures differ fundamentally. In our approach, a content-based message routing substrate is built to naturally enable task decoupling, dynamic reconfiguration, system monitoring, and run-time control. This is not addressed in the earlier work. Moreover, we present a proof-of-concept implementation and detailed performance results comparing distributed and centralized workflow management architectures, which is lacking in prior approaches.

A behavior preserving transformation of a centralized activity chart, representing a workflow, into an equivalent partitioned one is described in by Muth et al. [67] and realized in the MENTOR system [103]. The objective of the work is to enable the
parallel execution of the partitioned flow, while minimizing synchronization messages, and to analytically prove certain properties of the partitioned flow [67]. In a different set of transformations, parallelizing compiler-inspired techniques, including control flow and data flow analysis, are used to parallelize the business process to achieve the highest possible concurrency [71]. Both the transformation papers are complementary to our work since we operate with the original business process model without analyzing the process. An advantage of executing an unmodified process is that dynamic changes to the executing business process instances are possible, as their structure remains unchanged from the original specification.

Casati et al. present an approach to integrate existing business processes within a larger workflow [18]. They define event points in business processes where events can be received or sent. Events are filtered, correlated, and dispatched using a centralized publish/subscribe model. The interaction of existing business processes is synchronized by event communication. This is similar to our work in terms of allowing business processes to publish and subscribe. In our approach, activities in a business process are decoupled and are executed by activity agents, which are publish/subscribe clients, and the communication between agents is performed in a content-based publish/subscribe broker network.

In some Web applications, multiple instances of the application’s components may be dynamically provisioned over a set of machines in response to user requests [97, 89, 93, 95]. Typically, a data center environment is assumed in which complete knowledge about the available resources and application loads are known. For example, Tian et al. develop algorithms to load balance application load across a set of clustered servers and consider dependencies among application components [95]. This thesis, on the other hand, assumes a fully distributed architecture with no global knowledge of the applications, machines, or load, and hence the above algorithms cannot be readily applied.
2.2.4 Stream processing

There has been a lot of work on distributed stream processing engines in which a set of operators are installed in the network to process streams of data and execute SQL-like queries over the data streams [47, 2, 20, 81]. These operators input and output a set of streams and may filter, or change the data on these streams.

Borealis is a distributed stream processing engine in which streams are queried by a network of operators [2]. In addition to using a proprietary query language, Borealis does not support loops in the query network, which makes it unsuitable for general business process execution, specified in BPEL or similar languages, where looping constructs are commonplace.

In the IFLOW distributed stream processing engine, IFLOW nodes are organized in a cluster hierarchy, with nodes higher in the hierarchy assigned more responsibility [47]. For example, the root node is responsible for deploying the entire operator network to its children, and for monitoring the summarized execution statistics of this network. This is different from our completely distributed architecture in which brokers have equal responsibility.

While stream processing engines may bear some architectural resemblance to a set of agents executing a business process, there are issues related to business process execution that are not easily handled by stream processing engines. First, the stream processing work above is based on proprietary languages, not an industry standard such as BPEL. More significantly, a business process is conceptually not simply a data stream. There are notions of process instances and the accompanying state and isolation semantics that are not required in streams.

In addition to the semantic differences between processes and streams, process distribution in NIÑOS differs from the above work by exploiting an underlying content-based publish/subscribe system. As in IFLOW, our agents are decoupled by communicating using publish/subscribe content-based addresses instead of network identifiers. In ad-
dition, we utilize the *composite subscription* feature in PADRES to offload some of the agent processing to the publish/subscribe network. This simplifies the agents, and allows the publish/subscribe network to optimize this processing logic. This is further explained in Section 5.1.

### 2.2.5 Dynamic redeployment

While this thesis demonstrates the benefits of a distributed process execution architecture, it leaves open the problem of how a process should be deployed in order to satisfy certain goals. Ongoing work in this area attempts to dynamically redeploy the distributed business process based on certain the current workload, environment, and process goals expressed as formal service level agreements [22, 69]. It is important in this situation that the behavior of the process is unaffected by the redeployment of certain components. To support this capability, prior work has developed the notion of a transactional movement operation in which the redeployment of stateful publish/subscribe clients is guaranteed to satisfy a number of formal properties [36].

This thesis builds on our extensive prior work on building the PADRES system, a scalable, distributed publish/subscribe messaging middleware. The NIÑOS architecture exploits capabilities of the PADRES system including fine-grained content-based routing [29], and support for event correlations using composite subscriptions [54]. Other notable features of the PADRES system, such as user-tunable fault-tolerance [41], load-balancing [107], system policy management [104], historic data access [51], and the ability to route in cyclic overlay networks [56] are useful infrastructural properties for mission-critical enterprise applications.
2.3 Service management and monitoring

2.3.1 Service and resource discovery

This thesis presents a framework to enable decentralized discovery of both static and
dynamic service and resource attributes, and to enable the continuous monitoring of
resource updates. Some of the existing resource discovery approaches and techniques are
outlined in this section.

Many resource discovery schemes have been developed. This includes centralized dis-
cover based on indexing [82], a hierarchical indexing approach [27], Federated UDDI [86],
flooding-based discovery with Gnutella [1], and resource management using distributed
hash tables (DHT) [23]. A comprehensive survey of resource and service discovery in
large-scale multi-domain networks is given by Ahmed et al. who compare many promi-
nent discovery approaches ranging from industry solutions to state-of-the-art research [4].

Condor’s Matchmaker adopts a centralized architecture, where resource descriptions
and discovery requests are sent to a common central matching server that performs the
resource matching work [82]. This centralized approach is efficient for local area deploy-
ments for which Condor was initially designed. However, for large-scale decentralized
settings, the approach requires central administration and management for the operation
of the matchmaking server. While a central point of control eases administration, it is
also a central point of failure and a scalability bottleneck.

Globus’s MDS is a resource discovery approach based on a hierarchical architec-
ture [27]. In MDS-2, a Grid is comprised of multiple information sources that register
with index servers via a registration protocol. Resource requesters use a request protocol
to query directory servers to discover resource index servers and to obtain more detailed
resource descriptions from their information sources. The index servers form a hierar-
chical architecture. The top index server answers requests that discovery the resources
registered with its child index servers. This approach limits scalability, as requests trickle
through the root server, which easily becomes a bottleneck.

Federated UDDI consisting of multiple repositories that are synchronized periodically [86]. Federated UDDI is a popular and efficient solution for service discovery in distributed service networks. However, it is much too expensive to replicate frequently updated information, and, thus, it is hard to directly utilize this approach to support discovery of dynamic information.

Gnutella is an unstructured peer-to-peer network [1]. The discovery mechanism in Gnutella is based on flooding. Discovery requests are routed to all neighbor nodes of a given node. This happens until a timeout occurs or until the matched resources are retrieved. The flooding mechanism creates a large volume of traffic for networks with many nodes, connections and resources.

Alternatives have been developed where requests are more selectively propagated [37]. The proposed techniques include random walks, learning-based and best-neighbor-based propagation. Unlike in Gnutella, nodes choose their collaborating peer nodes based on expertise and preference. Resource requests are not flooded over the network, but directed to only a few selected nodes. Thus, the request-forwarding algorithms may not find all results for a request, if the matching resources are distributed among many different nodes.

Publish/Subscribe has been leveraged for service discovery [58]. In this approach, service attributes and discovery requests are translated to messages in publish/subscribe systems. This approach cannot support the combination of multiple discovery models, like the static, dynamic, and continuous models proposed in this thesis. Also, it does not consider the optimizations derived from processing multiple similar discovery requests. Li et al. [51] have studied historic data access in distributed publish/subscribe systems, which also use subscriptions as discovery messages. The objective of their work is different from ours and Li et al. do not address the resource discovery problem. Also, they do not develop optimizations for the processing of multiple concurrent discovery requests.
Approaches based on DHTs such as Chord [92] and Pastry [87] have been proposed. However, DHTs only efficiently support single keyword-based discoveries. A naive approach to resolving a range query issues separate point queries to nodes that correspond to each possible value within the query range is given in [23]. This approach becomes quite expensive for a typical sized range query and thus is unfeasible for more expressive resource requests. Techniques to more efficiently process range queries and multi-attribute queries in DHTs typically build additional indexes for the data items or add layers of indirection [33, 3, 68]. For example, Mercury assigns sets of nodes to be hubs for each resource attribute in the system [13]. These hubs index resources containing that attribute, and handle queries with that attribute. A limitation of the Mercury algorithm is that a multi-attribute query is decomposed into a set of single attribute queries that must be processed sequentially or in parallel.

The work in this thesis, on the other hand, utilizes a distributed publish/subscribe overlay that can efficiently evaluate multi-attribute and range constraints. Furthermore, the volatile message propagation paths in DHTs would prevent the similarity forwarding algorithms proposed in this thesis from being employed. Unlike typical peer-to-peer environments, however, this system is not designed for high node churn rates. It is also important to point out that we are not aware of any DHT-based resource discovery protocol that supports the continuous monitoring for resources with potentially dynamically updating attributes.

The problem of resource discovery complements our work on automatic service composition [35] where the resources, or services, that are found are automatically composed based on certain criteria to form a composite service with a specified interface. The automatic service composition work also leveraged the scalable matching capabilities of a distributed content-based publish/subscribe system as in this thesis. Therefore, the features proposed in this thesis such as the ability to continuously monitor for newly registered resources and the algorithms to exploit similarity among the resources and
discoveries can be used in service composition.

2.3.2 Service composition

This section presents the basic service composition problem and then follows with a discussion of how the work in this thesis relates to existing work in the field of automatic service composition and service computing.

Definition of service

A *service* $W$ is defined by input and output interfaces, $(W_{in}, W_{out})$, respectively. Each interface, in turn, consists of a set of *parameters*: $W_{in} = \{I_1, \ldots, I_m\}(m \geq 1)$ and $W_{out} = \{O_1, \ldots, O_n\}(n \geq 1)$. For a WSDL Web service, an interface may correspond to an individual operation, and the parameters of the input and output interfaces may correspond to the input and output message schemas of an operation. Likewise, for a service invoked by an RPC mechanism such as Java RMI, the set of argument types of a method would map to the input interface parameters, and the method return type to the output interface parameter. Figure 8.2 shows a few services with different number of input and output parameters. For example, the *DuplexPrinter* service has two input parameters of type *postscript* and *duplex_option*, and one output parameter that is of type *print_status*.

An invocation $R$ of a service is defined by the request parameters $R_{in} = \{I_1, \ldots, I_p\}(p \geq 1)$ and expected response parameters $R_{out} = \{O_1, \ldots, O_q\}(q \geq 1)$. Figure 8.2 illustrates three requests. For example, Request3 has two input parameters of type *html* and *duplex_option*, and one output parameter of type *print_status*.

Definition of successor

In a composition of services, a service $W'$ may *succeed* another service $W$ if the expected input parameters of $W'$ are contained within the set of output parameters from $W$:...
$W_{out} \supseteq W'_{in}$. Equivalently, it can be said that $W$ precedes $W'$.

This definition is extended to include the invocation $R$, such that service $W$ succeeds $R$ if $R_{in} \supseteq W_{in}$, and $R$ succeeds $W$ if $W_{out} \supseteq R_{out}$.

For example, for the services from Figure 8.2, service SimplePrinter succeeds service PDFConv, and Request1 precedes service SimplePrinter.

**Definition of compatibility**

Two services are said to be compatible if some of the input parameters of one is available as an output parameter of another. Formally, $W$ is compatible with $W'$ if $W_{out} \cap W'_{in} \neq \emptyset$. Likewise, an invocation $R$ is compatible with service $W$ if $R_{in} \cap W_{in} \neq \emptyset$ or $R_{out} \cap W_{out} \neq \emptyset$. Notice that the definition of compatibility is weaker than that of successors, and that a successor relationship implies compatibility.

From the examples in Figure 8.2, service PDFConv is compatible with (and precedes) service SimplePrinter, whereas the HTMLConv service is compatible with (but does not precede) service DuplexPrinter.

**Definition of atomic process**

It is possible that a single service $W$ can fully satisfy the input and output requirements of an invocation $R$. In this case, the “process” that fulfills $R$ is an atomic process consisting of the single service $W$.

Formally, $W$ satisfies $R$ if and only if

- $R_{in} \supseteq W_{in}$, and
- $W_{out} \supseteq R_{out}$.

That is, $R_{in}$ precedes $W$ which precedes $R_{out}$.

An atomic process that fulfills Request1 would consist of the single service SimplePrinter.
Definition of composite process

It is probably more common that a composition of services is required to satisfy an invocation’s input and output requirements.

A composition of services is represented as a directed cyclic graph $G$ in which the nodes are services, and edges denote compatibility relationships. More precisely, a directed edge from $W$ to $W'$ means that $W_{\text{out}} \cap W'_{\text{in}} \neq \emptyset$.

The directed graph in Figure 2.3 illustrates all the possible compatibility relationships among the services in Figure 8.2 and Request3.

![Compatibility graph](image)

Figure 2.3: Compatibility graph

The compatibility constraints of the above graph $G$ are not sufficient for the process defined by the graph to fulfill an invocation’s requirements. Instead, what is required is a subset of $G$ that is acyclic and where every service’s input parameters are fully satisfied by one or more predecessors. Formally, an invocation $R$ is fulfilled by a composite process represented by a directed acyclic graph (DAG) $D$ consisting of a sequence of services $\{W_1, W_2, \ldots, W_x\} (x \geq 2)$ if and only if

- $R_{\text{in}}$ precedes $W_{1\text{in}}$,
- $(R_{\text{in}} \cup W_{1\text{out}} \cup \ldots \cup W_{i-1\text{out}}) \supseteq W_{i\text{in}} (1 < i \leq x)$, and
- $(R_{\text{in}} \cup W_{1\text{out}} \cup \ldots \cup W_{x\text{out}}) \supseteq R_{\text{out}}$. 

There may be more than one DAG that fulfills an invocation request. For example, Figure 2.4 illustrates the two DAGs that result from the compatibility graph in Figure 2.3. In the first DAG in Figure 2.4, note that the DuplexPrinter service gets one of its inputs directly from the request, and another from the HTMLConv service. In the second DAG in Figure 2.4, the pdf output from the HTMLConv service is passed to the PDFConv service whose postscript output is then input to the DuplexPrinter service. While both processes in Figure 2.4 satisfy Request3, in this case, a user will most likely favor the first since it avoids an unnecessary invocation of the PDFConv service.

![Figure 2.4: DAGs resulting from Figure 2.3](image)

The two DAGs in Figure 2.4 are the results of the search for a process for invocation Request3. These two processes are returned to the user, who will decide which process to use. In this case, the first one is preferable since it avoids an unnecessary use of the PDFConv service.
Service modeling

Many models have been developed to facilitate automatic service composition. In one approach Web services are modeled using DAML-S and DAML+OIL, with a subset of DAML-S defined in first-order logic [72]. Other approaches model services using state transitions as in Roman [11] or Mealy machines in Colombo [10]. An interesting approach models services using Petri nets, and constructs a “service net” with input and output places corresponding to a service’s initial and final state, respectively [108]. In this thesis, services are modeled using publish/subscribe messages, mapping the task of constructing service relationships to publish/subscribe matching, and service composition partly to publish/subscribe routing. Future work will investigate the possibility of introducing a formal model to support validation and verification of a service composition.

Search techniques

The problem of searching for a composition of services is generally mapped to a planning or a digraph search problem [63]. For example, Bloom filters and A* algorithms can be used to search for chain structured service compositions [75]. An approach to integrate service discovery and automatic service composition has recently been proposed [49]. This integration is also supported by the work in this thesis: service discovery can be achieved by performing a process search with the TTL field set to one in order to retrieve only processes containing a single service. Amazon has presented an interesting application scenario for automatic service composition, where two services (Amazon’s e-commerce service and an external payment service) export complex interaction protocols and handle structured data in messages, making the composition of services with many interfaces complex and difficult [60]. Some work has also considered the performance characteristics of the services to compute compositions that optimize specified runtime optimization goals such as minimizing network traffic [40].

To the best of our knowledge, existing research only offer centralized architectures
for automatic service composition. This thesis argues for a potentially more scalable, efficient, and fault-tolerant distributed architecture that avoids single points of failure or bottleneck. A peer-to-peer architecture has been proposed in which the services, a request manager, a matching engine, and an objective engine are separated, but ultimately knowledge of service capabilities and service composition is still centralized [59]. The approach in this thesis, on the other hand, enables distributed matching and service composition search by using content-based publish/subscribe matching and routing.

Search result complexity

The results of an automatic service composition algorithm may range from a simple chain of services to arbitrary DAG structures, with the latter able to return more complex compositions. It has been noted that chain-only results may be “uncertain” if a service’s precondition can not uniquely determine a postcondition [100]. An approach to achieve a semantic Web service composition that can return DAG-like processes as results has been developed [46]. Such processes are also supported by this thesis.

Process execution

The processes found by an automatic service composition algorithm will eventually be executed by a centralized or distributed execution engine. The distributed execution of processes and workflows is an active area of research. Self-Serv [9] presents a peer-to-peer architecture that supports the distributed execution of service compositions using distributed routing in the peer-to-peer network. There have also been proposals for using a publish/subscribe system to achieve distributed process execution [47, 54]. However, none of these systems have considered automatic service composition in a distributed environment. Integrating distributed process search and execution in a unified architecture is an idea that will be explored in future work.
2.3.3 SLA modeling and monitoring

When a process is composed of third third party services, it is often desirable that both the service consumers and providers agree on the satisfactory level of service that the provider must fulfill and the service consumer expects. A Service Level Agreement (SLA) is introduced to address this requirement. It defines the quality of service that must be delivered by the service provider. The service consumer in turn relies on this level of service, as the service consumer might itself offer service guarantees to its consumers. The SLA also specifies the appropriate actions that must be taken if the terms and conditions are violated, such as the incurring of a penalty, for example.

There are a number of open standards available to model an SLA. For example, WSLA addresses the specification of SLA in a Web services environment [38]. WSLA defines the basic building blocks to model an SLA for Web services. In the specification, each SLA references the Web service as a whole. However, it can also refer compositions of multiple Web services [96]. It is composed of a number of metrics, which measure different aspects of a service. In the above example, the owner of the loan approval application might impose an SLA that every transaction must be completed in less than ten seconds. To enforce the SLA, metrics are created to measure the start and end time of the transaction. Metrics can also be defined by assembling simpler metrics to measure complexes scenarios. For example, after the start and end times are recorded, a metric for measuring the total transaction time can make use of the above metrics to compute the process duration.

The goal of the SLA is defined as Service Level Objective (SLO). It is a Boolean function expressed in terms of metrics. In the above example, the goal of the SLA on transaction time can be represented as an SLO in terms of the transaction time metric, in which the metric is compared with the agreed threshold. If an action must be executed when the SLO is violated, WSLA defines an Action Guarantee to specify the required actions.
In most cases, an SLA is verified during execution of a business process. However, it is also possible for an SLA to be verified at other stages. For example, an SLA to ensure that the service implementation is reviewed by a software architect should be verified during process deployment, since the requirement specifies that the software architect reviews the process before it is deployed for production.

Some researchers have focused on programmatically modeling an SLA. Sahai et al. [88] derive a set of constructs for modeling an SLA for Web services. In the proposal, elements such as SLA parameters and SLOs are introduced to model metrics and objectives respectively. Although some elements essential to SLA modeling are covered in this paper, other elements such as actions are not addressed in the proposed model. As a result, SLA developers are not able to execute an action upon SLA violation.

Keller and Ludwig [43] proposed different schematics for modeling SLAs. It is based on XML, which claims to provide some degree of extensibility. For instance, complex metrics are composed by a number of other metrics. In addition, SLOs can thus be defined in terms of metrics. This model poses some similarities with the proposed model in this thesis, but the latter is more extensible and flexible with the introduction of metric and SLO libraries. For example, new metrics are created by extending a metric type. New metric types can also be created in the same fashion. The hierarchical relationship between metric and SLO types allows a maximum degree of extensibility and reuse.

On the other hand, Lamana et al. [50] took a different approach in modeling SLA with the introduction of SLAng. It is an extension to existing business process languages. In SLAng, SLAs are defined in terms of a set of Quality of Service (QoS) parameters. These parameters are assigned to the target business process when it is being implemented. Executing these SLAs requires the target server to support these particular QoS parameters. The architecture becomes less extensible and flexible if new SLAs are introduced, because the server must be redesigned to support new QoS parameters.

A similar problem exists if the target SLA is implemented by a process. The runtime
architecture described in [88] involves creating monitoring agents to monitor the target business process. These agents instrument the business process by listening to its network usage. When it detects a change in the process, it executes another process which evaluates the SLA.

The above architecture requires the business process to update constantly to adapt to new SLA requirements. In addition, these monitoring agents pose huge performance overhead, because they actively listen to every event emitted from the business process. As a result, events that are not critical to the SLA execution are also being processed by the system. By contrast, the system developed in this thesis only accepts and processes events that are relevant to the SLA, thereby consuming fewer computing resources. In addition, it distributes the evaluation of an SLA among a set of monitoring artifacts, which can be run in a distributed environment. Its performance is generally better than approaches in which the SLA is monitored in a centralized manner [88].
Chapter 3

Transactional client movement

This chapter develops a primitive operation to allow publish/subscribe clients to transparently disconnect from one broker and reconnect to another. Sec. 3.1 formalizes transaction properties for mobile clients in a distributed content-based publish/subscribe system. Sec. 3.2 develops efficient protocols to support the above transaction properties, and presents a failure model and correctness proofs for the protocols under the stated failure conditions. Finally, in Sec. 3.3 the mobility protocols are implemented in a real system and their performance analyzed in detail with evaluations on a local testbed and a wide-area PlanetLab deployment.

3.1 Transaction properties

It is desirable for the movement of clients in a publish/subscribe system to be transparent to both the moving client and those it interacts with, such that an application consisting of stationary clients behaves the same as one where the clients move. Among other things, this means that clients should not miss any notifications while moving, and their movement should not be visible to others.

This section defines strong properties for client mobility in a publish/subscribe system similar to the relational database ACID [32] properties of Atomicity, Consistency,
Isolation, and Durability. We assume in-memory routing algorithms and hence will disregard the durability property. (Durability can be achieved by persisting state to stable storage.)

### 3.1.1 Movement operation

Consider Fig. 3.1 where Client A moves from broker $B_1$ to broker $B_7$ by issuing a `MOVE` command. The knowledge of where to move is application specific. For example, a virtual machine instance may wish to move to a less congested part of the network, or a stream processing operator may relocate to a machine with more memory.

The end result of a successful movement operation is that the client must sever its connection to broker $B_1$ and establish one with broker $B_7$ without losing any messages in the process. The movement may fail for any number of reasons including the target broker rejecting the moving client (perhaps because the broker is overloaded, or the client is not authorized to make the connection), in which case the client should remain connected to broker $B_1$, again with no loss of messages. The guaranteed transactional properties required of the operation are presented in the remainder of this section.

While movement can be achieved purely by managing a client’s connections, this chapter considers the more general case where a client’s execution location also moves. For example, a component managing a portion of a multiplayer game world may decide to migrate to a more optimal location in the network. While transferring the client...
computation and state, before the movement transaction has completed, there may be a copy of the client at both the source and target brokers ($B_1$ and $B_7$, respectively, in Fig. 3.1). This does not mean, however, that there are multiple functional instances of the client; the properties defined below ensure that only one copy of the client is “active” or visible to other clients in the system.

3.1.2 Publish/Subscribe system layers

A publish/subscribe system can be segmented into layers of well-defined functionality as depicted in Fig. 3.2. At a coarse grain, a client and broker interact with messages, including advertisements, subscriptions and publications from client to broker, and notifications from broker to client.

A client consists of application and publish/subscribe stub layers. The application layer encapsulates the application logic, be it an online game or a workflow management system. At this layer, transaction properties are domain specific but may rely on the properties of the lower layers. The publish/subscribe stub layer interfaces with a publish/subscribe broker. To support mobility, this layer must manage the phases of a moving client including queuing commands from the application, and retrieving notifications missed while moving. Since this chapter does not consider the application layer, the publish/subscribe stub layer in the client is referred to interchangeably with the client itself.
A publish/subscribe broker is divided into routing and messaging layers. The routing layer is concerned with how (un)advertisements, (un)subscriptions, and client movements influence the publish/subscribe routing tables, especially with routing state distributed across brokers. The messaging layer provides point-to-point communication between brokers. Messaging transactions, addressing concerns such as message ordering and synchronous versus asynchronous communication, are readily found in messaging products, such as MQSeries and JMS; they are not considered here.

In Fig. 3.2 messages from the client affect the broker routing tables, but notifications from the broker are passed to the application layer without modifying the publish/subscribe stub state. However, since notifications are exposed to the application, it is necessary to define guaranteed properties for the notifications delivered to a client. The remainder of this section defines properties on the mobile client state (Sec. 3.1.3), the notifications delivered by the routing layer (Sec. 3.1.4), and the distributed routing table state (Sec. 3.1.5). Algorithms that satisfy the properties in each layer are presented, along with proofs, in Sec. 3.2.2, 3.2.3, and 3.2.4, respectively.

3.1.3 Client layer

We first focus on the correctness of a client movement protocol by way of properties on the state of clients. Each client must be in a unique state at all times. In standard publish/subscribe systems, a client may be in a connected or disconnected state. As we shall see in Sec. 3.2.2, in a system that supports mobility, more states are required.

Atomicity: We define two atomicity properties:

(a) After the transaction completes, a moving client must be either at its source or target broker, but not both.

(b) For a transaction consisting of a sequence of operations \( \{o_1, ..., o_n\} \) that transition a client from state \( s_0 \) to \( \{s_1, ..., s_n\} \), the final state of the client must be \( s_0 \) or \( s_n \). That is, either all operations are completed, or none are.


**Consistency:** There must be at most one running instance of each client. This states that a movement should not result in two instances of a client, with one instance at the source broker and another at the target.

**Isolation:** Suppose it is possible for an operation by a client to observe the state of another client. Given a transaction $T_x = \{o_1, ..., o_n\}$ that causes a client to change its state from $s_0$ to $\{s_1, ..., s_n\}$, an operation in another transaction $T_y$ should only observe $s_0$ or $s_n$.

The properties above make a movement operation transparent in terms of the client’s state. For example, if isolation is violated, then one of the intermediate states of a moving client may be observable, and the movement is no longer transparent. This point will become clearer in Sec. 3.2.2 where the state transitions of a moving client are described.

### 3.1.4 Notifications

We now define the properties required of the notifications delivered by the routing layer to the publish/subscribe clients. Let $P_i(t_a \rightarrow t_b)$ denote publications issued by client $i$ in the time interval bounded by $t_a$ and $t_b$, and $N_i(\cdot)$ refer to notifications received by client $i$. If necessary, notifications received by the copy of the client at the source or target broker are distinguished as $N^S_i(\cdot)$ or $N^T_i(\cdot)$, respectively.

**Atomicity:** Notifications—the delivery of publications to interested subscribers—are atomic. In the case of interested stationary clients, they are delivered exactly once; or in the case of interested moving clients, delivered exactly once to either the source or target client copy, but not both.

**Consistency:** Consider a client that initiates a movement operation at time $t_0$ in Fig. 3.3. The notifications $N^-(t_0 \rightarrow t_\infty)$ received by a client if it does not attempt to move should be the same as those $N^T(t_0 \rightarrow t_\infty)$ received if it successfully moves to the target, and the same as those $N^S(t_0 \rightarrow t_\infty)$ received if the movement fails and it remains at the source. (Time $t_\infty$ is when the client permanently disconnects from the publish/subscribe
network, so $N^S(t_0 \rightarrow t_\infty)$ contains the set of notifications the client will ever receive starting from time $t_0$.) Notice that the location of a client may affect the order of notifications, which is why this property only requires that notifications are eventually delivered to the client.

**Isolation:** Consider a client $C_i$ that initiates a movement operation at time $t_0$, and assume that the publications $P_i(t_0 \rightarrow t_\infty)$ issued by $C_i$ if the movement is successful, are the same as the publications $P'_i(t_0 \rightarrow t_\infty)$ it issues if the movement fails. The notification received by every other client $C_j \neq C_i$ must be the same whether the movement completed or not: $N_j(t_0 \rightarrow t_\infty) = N'_j(t_0 \rightarrow t_\infty)$.

The notification properties above are designed to make a client’s movement transparent, in terms of the notifications delivered both to itself and other clients in the network. For example, if atomicity is violated, a message may be processed twice by a client (once each by the copies of the client at the source and target brokers). The effects of this double processing of a message can be observed, making the movement visible, and thus may violate isolation.

### 3.1.5 Routing layer

Finally, we define the properties of the routing table state in a distributed content-based routing protocol.

**Atomicity:** For an operation $o$ (such as advertise, subscribe, publish, or move) by a client, a publish/subscribe protocol defines routing table updates $U(o)$ that should occur. To be atomic, either all updates in $U(o)$ occur, or none.

**Consistency:** For every advertisement $A$ that matches a subscription $S$ issued by a client $C$, it must be that:

(i) At every broker $B_i$ in the path from $\text{publisher}(A)$ to $C$: $S.lasthop \neq A.lasthop \land S.nexthop = A.lasthop$.

(ii) At every broker $B_i : B_i$ and $A.lasthop$ are neighbours in the path from $B_i$ to
These properties define the minimal set of routing table entries required to deliver notifications to all interested subscribers. Note that a routing table may have additional (perhaps stale) entries and still be considered consistent.

**Isolation:** Consider the advertisements $A_i$ and subscriptions $S_i$ issued by a moving client. Let $RT_B$ be the routing table entries at broker $B$, and $RT_B(A_i)$ and $RT_B(S_i)$ be the subset of $RT_B$ corresponding to $A_i$ and $S_i$, respectively. To satisfy isolation, for every broker $B : [RT_B - RT_B(A_i, S_i)]_{beforemove} = [RT_B - RT_B(A_i, S_i)]_{aftermove}$. Informally, a movement should only update routing entries of the advertisements and subscriptions issued by the moving client; other clients’ routing state should be unaffected.

The above properties are satisfied by well-known publish/subscribe routing protocols under non-failure conditions, and it is not difficult to adapt these protocols to tolerate failures in the case where faults are not permanent: brokers that crash are eventually restarted or replaced with a working broker, and link failures never permanently partition any set of brokers.

Under such failures, a publish/subscribe protocol can be made fault-tolerant by persisting the algorithmic and queue state of each broker, to recover from node and link failures, respectively. The algorithmic state—data managed by publish/subscribe protocols such as advertisements and subscriptions—is typically kept in memory, but can be persisted. The queue state includes unprocessed incoming messages at a broker and undelivered outgoing messages. The reliable delivery of these messages between brokers can be achieved using persistent queues. One implication of recovering broker failures locally without coordination among brokers, is that the recovery of a crashed broker may take arbitrarily long (perhaps requiring a manual restart), and so message delays may be unbounded, although eventual delivery is guaranteed.

The fault tolerance scheme above is straightforward and uses well-known technologies, and serves to demonstrate it is possible to implement a distributed publish/subscribe
routing protocol that satisfies the atomicity, consistency, and isolation properties described above. Research on more sophisticated fault-tolerant algorithms is ongoing [41, 79] and the above properties can serve as a guide for such research. They are also used in this chapter as a basis for the proofs of subsequent properties. In this way it is clear what properties are required of a robust publish/subscribe routing protocol in order to support the higher-level properties we address more fully below.

3.2 Client movement protocol

This section presents client movement protocols and proves they satisfy the transactional properties in Sec. 3.1. Unlike typical publish/subscribe mobility [25], where the client disconnects from one broker and reconnects to another, these protocols provide strong transactional movement guarantees and are more efficient, as we show in Sec. 3.3.

3.2.1 System model

We assume an acyclic overlay of publish/subscribe brokers that satisfy the properties in Sec. 3.1.5. Notably, node crashes or network faults in the publish/subscribe layer are masked by the routing protocols. A method to adapt existing publish/subscribe routing algorithms to be fault-tolerant was sketched in Sec. 3.1.5.

A mobile container associated with each broker encapsulates a coordinator (to execute the movement protocol) and the clients themselves. In this way, the middleware has full control over the deployment of clients. Also, we assume that the components within a container do no individually fail, so a crash failure of a coordinator implies a failure of the associated clients and vice versa.

Our movement algorithms are based on the three-phase commit (3PC) distributed transaction protocol [90].\(^1\) As such, we can inherit 3PC’s failure model. In particular,

\(^1\)Unlike 2PC, 3PC supports non-blocking transactions, and nicely conforms to the message exchanges
crash failures of mobile clients and coordinators are allowed, and two network failure models are supported: (i) the network delivers messages within a bounded delay, in which case the non-blocking 3PC is used and movement transactions are guaranteed to complete within a bounded time; or (ii) message delays are unbounded in which case we use a blocking variant of the protocol and movement may block.

### 3.2.2 Client layer

We present a movement protocol that satisfies the properties in Sec. 3.1.3. It consists of a conversation between the source broker a client is moving from and the target broker it is moving to as outlined in Fig. 3.3. While the protocol is handled by the source and target brokers, the brokers make use of a reconfiguration message (message (2) in Fig. 3.3) that is processed by all brokers along the path. The handling of this message is detailed in Sec. 3.2.4. The movement protocol modifies the state of the client and that of the source and target brokers as summarized in Fig. 3.4. The protocol proceeds as follows when a client moves from source broker $B_i$ to target broker $B_j$:

First, $B_i$ sends message (1) to $B_j$ with data about the moving client such as its ID, and its subscriptions and advertisements.

If $B_j$ decides to accept the client, $B_j$ initializes a transaction state for the new client,
and then issues message (2) containing the client id and its subscriptions and advertisements. Message (2) executes routing table reconfiguration as described in Sec. 3.2.4. If $B_j$ does not accept the client it sends a reject message (3) to $B_i$.

If $B_i$ gets message (2), it stops the client, and sends message (4) to $B_j$, along with any queued publications for the client. Otherwise $B_i$ receives message (3) and resumes the client.

$B_j$ receives message (4), and dispatches it to the new client, which merges the notifications in the payload of this message with those in the queue at the target node. $B_j$ also sends message (5) to $B_i$, upon receipt of which $B_i$ finally cleans up any state associated with the client.

The client and coordinator at the source and target sites, all of which participate in the protocol, are modelled in Fig. 3.4. State transitions are labelled by the input transition trigger message and the generated output message. Messages between the application, mobile client, and coordinator are marked as indicated in the legend in Fig. 3.4.
The *global state* of a distributed protocol is defined by a vector of local states and outstanding messages in the network, and the global state transitions when a local state transition occurs. Transitions between global states create a reachable global state graph. For example, a possible global state is one where the source and target coordinators are in the *init* state with the *move* message (sent by the client to the coordinator) outstanding in the network. When this message is received, the global state transitions to one where the source coordinator moves to the *wait* state, the target coordinator remains in the *init* state, and the *negotiate* message (sent from the source to target coordinator) is in the network. Fig. 3.5 is the global reachable state graph for the local coordinator state graphs in Fig. 3.4. Note that the initials of the local coordinator state names are used to label the global states in Fig. 3.5.

The table in Fig. 3.4 lists possible concurrent client and coordinator states. For example, when both coordinators commit the transaction, the source and target clients must be in the *clean* and *started* states, respectively. From the table in Fig. 3.4, we...
see that for any global state in Fig. 3.5, two properties hold: (1) in a final global state, exactly one client is\textit{started} and the other is\textit{clean}; and (2) in any intermediate global state, at most one client is\textit{started}.

Using these properties, we can prove the client state properties in Sec. 3.1.3.

**Atomicity Proof:** (a) It follows from property (1) that a client only exists in the source or target broker after a movement. (b) Since we assume the client and coordinator experience the same faults (because they run on the same node), and the coordinator is guaranteed to abort or commit (barring an unrecoverable crash failure), the client will, according to Fig. 3.5, end up in its initial state (if the coordinator aborts), or the final state (if it commits).

**Consistency Proof:** It follows from property (2) that there is at most one running instance of a client.

**Isolation Proof:** A client’s state can only be inferred by notifications received from it, in which case the client is in the running state. Since a client is never running until the movement has committed or aborted, it is not possible to observe an intermediate state of a moving client.

### 3.2.3 Notifications

We now prove the notification properties from Sec. 3.1.4, assuming the use of a publish/subscribe layer that satisfies the routing state properties in Sec. 3.1.5.

**Atomicity Proof:** Correct routing state properties (namely consistency and atomicity) will ensure that the routing tables are configured so as to deliver notifications to stationary clients. For mobile clients, in any committed global final state in Fig. 3.5, the reconfiguration message would have been sent resulting in the appropriate updates to the routing state. Hence, a correct routing layer will deliver all messages to either the source or target client.

**Consistency Proof:** Based on the protocol in Fig. 3.3, we wish to show that the
set of notifications \( N^S(t_0 \rightarrow t_{\infty}) \) received had the client not moved is equivalent to the notifications \( N^T(t_0 \rightarrow t_{\infty}) \) if it does move. We first note that \( N^T(t_0 \rightarrow t_{\infty}) = N^T(t_2 \rightarrow t_{\infty}) \cup N^S(t_0 \rightarrow t_3) \) since the target receives the latter messages from the source as part of the protocol. And, by definition, \( N^S(t_0 \rightarrow t_{\infty}) = N^S(t_0 \rightarrow t_3) \cup N^S(t_3 \rightarrow t_{\infty}) \).

It suffices to show that \( N^T(t_2 \rightarrow t_{\infty}) \) and \( N^S(t_3 \rightarrow t_{\infty}) \) are equivalent. Note that only at \( t_3 \) have routing table entries for the target client been properly updated. Assuming routing states are setup correctly by the routing table layer, every publication \( m \in N^T(t_2 \rightarrow t_{\infty}) \) also belongs to \( N^S(t_3 \rightarrow t_{\infty}) \). Similarly, by virtue of an acyclic topology in which messages are processed in order, every publication \( m \in N^S(t_3 \rightarrow t_{\infty}) \) also belongs to \( N^T(t_2 \rightarrow t_{\infty}) \). Hence, \( N^T(t_2 \rightarrow t_{\infty}) \) and \( N^S(t_3 \rightarrow t_{\infty}) \) are equivalent.

Also, even if the movement aborts, no routing table entries are modified that affect the source broker, so \( N^S(t_0 \rightarrow t_{\infty}) \) equals \( N^-(t_0 \rightarrow t_{\infty}) \) which are the notifications the client receives if it does not attempt to move.

**Isolation Proof:** We assume a client \( C_i \) issues the same publications regardless of its location: \( P_i(t_0 \rightarrow t_{\infty}) = P'_i(t_0 \rightarrow t_{\infty}) \). It remains to show each publication is issued exactly once. A client may only publish while in a running state. We have seen that a client is never in a running state at both the source and target, and that after the transaction completes, it will be in the started state at the source or target, but not both. Therefore, a client cannot issue a publication from both the source and target, and since publications are buffered (not dropped) while in a non-started state each publication is issued exactly once.

### 3.2.4 Routing layer

We now outline the routing layer protocol to achieve client movement according to the transactional properties defined in Sec. 3.1.5. The objective of the routing reconfiguration algorithm is to efficiently maintain valid routing configuration states during client movement.
In traditional client movement [25, 70], a client disconnects from its source broker after unadvertising and unsubscribing its history, and these messages propagate through the network. Then, the client connects to the target broker and reissues its advertisements and subscriptions, which again propagate. This is expensive, especially since (un)advertisements are flooded. The protocol can be improved by enabling advertisement and subscription covering, where (un)advertisements and (un)subscriptions are quenched by covering advertisements and subscriptions.

Contrary to traditional assumptions, with frequent mobility, enabling covering may actually degrade performance. Consider publisher pub₁ (pub₂) issuing advertisement adv₁ (adv₂) and let adv₂ cover adv₁. Suppose adv₁ is sent first, and flooded. Then, adv₂ is issued and also flooded. With advertisement covering, there are links where it is redundant to send both adv₁ and adv₂. For example, when a broker B’ forwards adv₂ to broker B'”, it will also issue an unadvertisement for the previously sent adv₁ since adv₂ covers adv₁. It turns out that adv₁ will have to be unadvertised over all links not on the path between pub₁ and pub₂. So, we have a situation where both adv₁ and adv₂ were flooded, and adv₁ was unadvertised throughout most of the network, which is more expensive than if covering is not enabled. Such situations are more likely when clients move frequently, and issue many advertisements and subscriptions.

We develop a routing reconfiguration protocol to achieve the best-case efficiencies of the traditional covering-based mobility algorithm, and not suffer from its pathological deficiencies. The algorithm reconfigures the routing table hop-by-hop along the path between source and target brokers.

Since the overlay is acyclic, there is only one route from a source broker $B_i$ to a target broker $B_j$ (assume $i < j$). This route, $RouteS2T$, is a sequence of brokers, $\langle B_i, B_{i+1}, \ldots, B_j \rangle$, such that $(B_m, B_{m+1})$ is an edge in the network, where $i \leq m < j$. The predecessor and successor of $B$ in $RouteS2T$ are denoted as $RouteS2T.pre(B)$ and $RouteS2T.suc(B)$, respectively.
Suppose a publisher at broker $B_i$ moves to broker $B_j$ ($i < j$). After movement, advertisement $adv$ at the old publisher becomes $adv'$ at the new publisher.

**Claim 1**: The routing configurations of $adv$ and $adv'$ are identical at all brokers except for brokers $B \in \text{RouteS2T}$.

**Proof**: Consider a broker $B \notin \text{RouteS2T}$. $B$ must have zero or one neighbours $B_n \in \text{RouteS2T}$ (since the network is acyclic). In the former case, $B$ must have a neighbour $B'_n$ that is on the path to both $B_i$ and $B_j$ (since the topology is acyclic). In the latter case, the only path to $B$ from either $B_i$ or $B_j$ is through $B_n$. Hence, in either case the routing configuration at $B$ consists of an entry for $adv$ or $adv'$ from either $B_n$ or $B'_n$.

**Claim 2**: The routing configurations of $adv$ and $adv'$ at all brokers $B_i \in \text{RouteS2T}$ are different.

**Proof**: Consider a broker $B \in \text{RouteS2T}$ and $B$ is neither $B_i$ nor $B_j$. Since $B_i$ and $B_j$ are at opposite ends of $\text{RouteS2T}$, the neighbour from which $B$ receives $adv$ (sent by $B_i$) must be different from which it receives $adv'$ (sent by $B_j$), and hence the routing configuration of $B$ is different in the two cases. Likewise, the routing configuration at $B_i$ and $B_j$ will be different.

This means that only the routing state at brokers along the route from target to source broker need to be modified. It is possible to simulate un-advertisement and re-advertisement by modifying the routing configuration $rc(adv)$ to be $rc(adv')$ hop-by-hop from $B_i$ to $B_j$ (source to target) or from $B_j$ to $B_i$ (target to source). Since the information about an advertisement at a broker influences the routing of matched subscriptions, both the records in the $SRT$ and the records in the $PRT$ must be modified.

To move a publisher that has issued an advertisement $adv$, the routing configuration at every broker $B_i \in \text{RouteS2T}$ is modified as follows. The records $(adv, \text{RouteS2T.pre}(B_i))$ in the $SRT$ are modified to $(adv', \text{RouteS2T.suc}(B_i))$. For the records $(sub, \text{lasthop})$ in the $PRT$ where $sub$ intersects $adv$, there are three cases to consider: (1) $sub.\text{lasthop} =$
\( B_x \notin \text{RouteS2T} \); (2) \( \text{sub}.\text{lasthop} = \text{RouteS2T}.\text{suc}(B_i) \); and (3) \( \text{sub}.\text{lasthop} = \text{RouteS2T}.\text{pre}(B_i) \)

For the first case, since \( \text{adv}.\text{lasthop} = \text{adv}'.\text{lasthop} \) in broker \( B_x \), any subscription \( \text{sub} \) intersecting \( \text{adv} \) also intersects \( \text{adv}' \), and will be forwarded to \( B_i \). If \( \text{sub} \) has not already been forwarded to \( \text{RouteS2T}.\text{suc}(B_i) \), it also needs to be forwarded to \( \text{adv}'.\text{lasthop} \). Note that at \( B_i \), \( \text{adv}'.\text{lasthop} = \text{RouteS2T}.\text{suc}(B_i) \).

For the second case, \( \text{sub}.\text{lasthop} = \text{adv}'.\text{lasthop} \), which means that \( \text{sub} \notin \text{rc}(\text{adv}') \). Unless \( \text{sub} \) intersects an advertisement besides \( \text{adv} \), it is removed from the PRT.

For the third case, \( \text{sub}.\text{lasthop} = \text{adv}.\text{lasthop} \), which means that \( \text{sub} \notin \text{rc}(\text{adv}) \), so it must also match some other advertisement \( \text{adv}_1 \) where \( \text{adv}_1.\text{lasthop} = \text{RouteS2T}.\text{suc}(B_i) \) or \( \text{adv}_1.\text{lasthop} \notin \text{RouteS2T} \). If \( \text{sub} \) has not already been forwarded to \( \text{RouteS2T}.\text{suc}(B_i) \), it needs to be forwarded there.

To guarantee atomicity of the movement transaction, it is necessary to construct a copy of \( \text{rc}(\text{adv}') \), which is the revised version of \( \text{rc}(\text{adv}) \), at each broker along \( \text{RouteS2T} \). Because of Claim 1, the algorithm only needs to revise the routing configurations along \( \text{RouteS2T} \). If the transaction commits, the old routing configuration \( \text{rc}(\text{adv}) \) is deleted hop-by-hop, otherwise, \( \text{rc}(\text{adv}') \) is deleted hop-by-hop.

### 3.3 Evaluation

The main conclusion of the evaluations is that the reconfiguration protocols exhibit more stable performance than the traditional covering-based movement protocol. The covering protocol’s performance varies greatly and is more susceptible to pathological scenarios.

The protocols in this chapter are implemented in the Java-based PADRES content-based publish/subscribe prototype. Experiments performed on a cluster of 1.86 GHz machines with 4 GB of RAM that mimics an enterprise data centre environment and offers a controlled system for meaningful analysis. Evaluations are also conducted on heterogeneous nodes in the wide-area PlanetLab testbed, representing a geographically
distributed system administered by one or more enterprises, and also serves to stress the protocols under the unpredictable and shared PlanetLab network.

By default the 14 broker overlay in Fig. 3.6 is used with each broker running on a separate machine. A mobile coordinator is co-located with each broker allowing any broker to host mobile clients, and so we will not distinguish between brokers and mobile coordinators. Unless otherwise stated, clients connect to a random broker at startup, and then initiate their movement pattern, pausing for ten seconds at each broker between movements. Each subscriber is assigned a subscription randomly from the subscription workload.

Metrics include network traffic, movement duration and movement throughput. Network traffic, measured as the sum of messages transmitted over each overlay link, includes publications, (un)subscriptions, and (un)advertisements. Assuming roughly equal message sizes we approximate network traffic with message counts. Movement duration is the time to complete a client movement transaction, and movement throughput measures the number of movement transactions the system can process in a given time.
Each client issues a subscription chosen from the four subscription workloads in
Fig. 3.7, where covering relationships are shown. For example, in Fig. 3.7(a), the root
subscription covers all the others. Not shown is a Random workload in which subscrip-
tions from all four workloads in Fig. 3.7 are selected uniformly. The details of individual
subscriptions are omitted since it is primarily the covering relationships among them that
affect the results.

**Subscription Workload:** We evaluate the sensitivity of the protocols to different
subscription workloads. We first consider an experiment in which 400 clients, initially
connected to Brokers 1 and 2, each repeatedly performs a movement between Brokers 1
and 13, and 2 and 14, waiting for ten seconds at each broker before moving again.

The movement latency is shown in Fig. 3.8 for the reconfiguration and covering move-
ment protocols. Each point in the plot represents the time to complete the protocol
(vertical axis) for a movement that starts at a given time (horizontal axis). Notice that
the reconfiguration protocol is more than an order of magnitude faster than the covering
one. Also observe that movements at the beginning of the experiment take longer than
those at the end due to the load imposed by joining clients. To avoid skewing steady
state performance, we ignore this setup phase in subsequent results.

In Fig. 3.8(b), we see that clients moving between Brokers 1 and 13 are slower than
those between Brokers 2 and 14. This is because odd valued subscriptions as numbered in Fig. 3.7 are initially assigned to Broker 1, and even ones to Broker 2. Since subscription 1 in the covered and tree workloads (Figs. 3.7(a) and 3.7(c)) cover more subscriptions than others, it is more likely to cause a pathological subscription propagation during movement as explained in Sec. 3.2.4. We confirm this with results that show there is almost no variance in movement latencies with the chained workload (where each subscription covers at most one other), and that the variance increases with the covered workload (where the root subscription covers the remaining nine).
Fig. 3.9(a) summarizes the latencies for different workloads, with the $x$-axis values corresponding to the number of covered subscriptions in the workload. The reconfiguration protocol exhibits little variation in latency, while the covering protocol performs worse—almost two orders of magnitude in the worst case—when more covering is present in the subscription workload.

Fig. 3.9(b) shows the number of messages normalized to the number of movements that occur during the experiment. Since clients keep moving during the experiment, a slow protocol will result in fewer movements, and so the per-movement message counts...
Chapter 3. Transactional client movement

(a) Movement latency

Figure 3.13: Topology size

are a more fair representation of a protocol’s message overhead. The normalized message values are plotted as lines with values on the left vertical axis, and for completeness, the number of movements are plotted as impulses with values on the right vertical axis. First observe in Fig. 3.9(b) that the reconfiguration protocol maintains a stable message overhead regardless of workload, and is able to complete roughly the same number of client movements during the experimental duration. On the other hand, the covering protocol performs fewer movements with the tree and covered workloads, a direct consequence of each movement taking longer to complete as we saw in Fig. 3.9(a).

It seems odd that, compared to the tree workload, the covered workload imposes less per-movement message overhead as seen in Fig. 3.9(b) but results in a longer movement latency. This apparent discrepancy is due to an underlying bimodal behaviour of the covering algorithm with the covered workload. It is cheap for the covering protocol to move one of the non-root subscriptions in the covered workload (subscriptions 2 to 10 in Fig. 3.7(a)) because their propagation is quenched by the presence of the covering root subscription (subscription 1 in Fig. 3.7(a)). Moving the root subscription, however, is expensive, since it triggers the propagation of all the non-root subscriptions. The movement of the root subscriptions occurs seldom relative to the non-root ones which is why the message overhead of the covered workload is less than that of the tree one which
contains fewer non-leaf subscriptions. However, when the root subscription does move, it causes a burst of messages that causes significant congestion and has a large impact on movement latencies.

We emphasize that unlike the covering protocol, the reconfiguration protocol’s message load and latency results are stable with respect to subscription workloads.

**Number of Clients:** We evaluate scalability by varying the number of moving clients from 400 to 1000. Fig. 3.10(a) illustrates two important points: the reconfiguration algorithm performs much better than the covering algorithm, and the latter’s performance degrades with more clients, while the former maintains stable performance. We see the same stable message overhead for the reconfiguration protocol in Fig. 3.10(b). There is an apparent paradox with the reconfiguration protocol achieving faster movement despite more total messages (which is the product of the normalized overhead and the number of movements). This occurs because it is able to isolate these messages within the path between source and target brokers, and because it does not suffer from bursty propagation of messages that congest the network. This also explains why a slight increase in the covering protocol’s message overhead dramatically impacts the latency results in Fig. 3.10(a).

**Single Client:** This experiment isolates the effects of moving a single subscription in the covered workload with 400 clients. Only the root subscription (subscription 1 in Fig. 3.7(a)) is moved. Fig. 3.11 shows that the covering protocol has much worse movement latency and message load. Since only the root subscription moves here, we confirm the reason for this is precisely due to the pathological case for the covering protocol, where subscriptions (unsubscriptions) of the root subscription induce unsubscriptions (subscriptions) of the non-root subscriptions.

**Incremental Movement:** This time, we keep the number of clients constant at 400, and increase the number of these that move. Fig. 3.12(a) again shows the superior and stable latencies of the reconfiguration protocol. The covering protocol exhibits an
interesting behaviour. In the experiment each increment of ten moving subscriptions are successively chosen as follows: ten covering (i.e., root) subscriptions from the covered workload, ten covering from the tree workload, ten covering from the chained workload, ten covered (i.e., leaf) chosen randomly from the previous three workloads, and finally ten from the distinct workload. Notice that the first four sets of subscriptions have less and less covering, with the last two sets not covering any. For example, the first ten are chosen only from the covering workload (which has the most covering). And so, we expect the incremental effect of moving ten tree workload subscriptions to be greater than that of ten chained subscriptions. Indeed, we observe in Fig. 3.12(a) that the slope of the covering protocol’s latency between the first ten and twenty (tree workload) moving clients is slightly steeper than for the next ten (chained workload). The movement of
subscriptions that do not cover any others, introduced when forty clients move, can be performed very quickly and contributes to a reduction in the average latency of the covering protocol. Likewise, the last ten subscriptions do not cover any, and they further reduce the average latency of the covering protocol. Fig. 3.12(b) also shows how the message overhead for the covering protocol decreases when moving subscriptions with less covering. The results in Fig. 3.12 nicely illustrate how the covering relationships of the subscription workload affect the performance of the covering protocol.

**Topology Size:** In this experiment we increase the number of brokers in the topology but keep the path length between source and target brokers constant by only moving clients between Brokers 1 and 12, and Brokers 2 and 14. The covered workload is used here to try to induce an exaggerated effect. Fig. 3.13 shows that increasing the topology size affects neither the latency nor message load drastically. This is expected since the reconfiguration protocol sends messages between the source and target brokers, and the covering protocol is primarily affected by congestion in the path between these two brokers. However, we wish to note that if there were clients moving in other parts of the network, they would be affected by the covering protocol but not the reconfiguration protocol.

**Wide-area PlanetLab deployment:** Evaluations on PlanetLab confirm the trends observed in the local testbed, but show longer movement latencies due to the more limited network and compute resources on PlanetLab. For example, similar to the experiment in Fig. 3.8, Figs. 3.14(a) and 3.14(b) now show the results of a 14 broker topology with 100 moving clients in the wide-area testbed. The reconfiguration protocol performs movements faster than the covering algorithm, but both take longer than in the local environment. Also the latencies vary more due to the unpredictable resource availability in the shared PlanetLab environment. As well, the trends in the local testbed in Fig. 3.9 occur in the wide-area deployment: Fig. 3.14(c) shows the covering protocol suffers with workloads with more covering, and Fig. 3.14(d) confirms the reconfiguration
protocol imposes a smaller message overhead than the covering protocol, and completes movement transactions at a faster rate. Other experiments on PlanetLab reinforce earlier conclusions such as the insensitivity of either protocol to the size of the network topology.
Chapter 4

Incremental filter aggregation

This chapter develops an algorithm to manage the potential congestion triggered by unsubscription messages. Sec. 4.1 formalizes properties of subscription propagation in content-based routing networks, and defines a metric to capture the similarity between subscriptions. Sec. 4.2 devises a subscription packing algorithm to exploit the relationships among triggered subscriptions in order to optimize the processing of these subscriptions. The algorithm builds on a simplified data structure for maintaining subscription covering relationships. Sec. 4.3 uses the similarity metric and subscription propagation properties above to develop a distributed algorithm to incrementally prune the propagation tree of a subscription that is to be removed. Sec. 4.4 then presents a detailed evaluation of the performance of an implementation of the distributed pruning algorithm.

4.1 Subscription propagation properties

We first analyze the basic issues related to subscription propagation in an acyclic content-based routing overlay, and then we identify and define a measure of similarity between subscriptions. These properties are fundamental to the subscription pruning algorithm developed in Sec. 4.3.
4.1.1 Nature of subscription propagation

In this section, we describe certain properties of how subscriptions are propagated in traditional acyclic content-based routing overlay networks.

In the following discussion, the publisher host broker (PHB) refers to the broker to which a publisher connects. Similarly, the subscriber host broker (SHB) is the broker to which a subscriber connects. Note that these are only logical designations, and any given broker may play the role of both a PHB and SHB.

Consider two types of subscriptions: root subscriptions which are not covered by any other subscription in the system and non-root subscriptions which are covered by some subscription in the system. It turns out that the propagation paths of each class of subscriptions in an acyclic topology follows following properties.

Property 1. Without the covering optimization, the propagation of any subscription \( S \) is a tree.

Proof. Based on the subscription routing protocols [8], \( S \) is disseminated from the subscriber to the host brokers of publishers with intersecting advertisements. In this way a path is constructed between each intersecting publisher’s PHB and the subscriber’s SHB.
as required for reverse path forwarding. The dissemination of \( S \) is a tree rooted at the SHB of \( S \) with leaves at the potentially interesting publishers’ PHB.

For the next property, let \( S_r \) and \( S_n \) be a root and non-root subscription, respectively, such that \( S_r \) covers \( S_n \): \( S_r \supset S_n \). Also, let \( \{A_{S_r}\} \) and \( \{A_{S_n}\} \) denote the set of advertisements that match \( S_r \) and \( S_n \), and let \( T_{S_r} \) and \( T_{S_n} \) denote the subscription propagation trees of \( S_r \) and \( S_n \).

**Property 2.** With the covering optimization, \( T_{S_r} \) remains the same, and \( T_{S_n} \) can be reduced to a linear path.

*Proof.* First, we note that the propagation of a root subscription \( S_r \) is not quenched by any other subscription since nothing else covers it, so its propagation is unaffected. It remains to show that the propagation of a non-root subscription \( S_n \) is a path. The argument proceeds as follows:

(1) From the definition of subscription covering, the advertisements that match \( S_n \) also match \( S_r \): \( \{A_{S_r}\} \supset \{A_{S_n}\} \).

(2) Then, based on the content-based routing algorithms, for each advertisement \( A_i \in \{A_{S_n}\} \), there exists a path from the SHB of \( S_r \) to the PHB of \( A_i \) in \( T_{S_r} \), and also a path from the SHB of \( S_n \) to the PHB of \( A_i \) in \( T_{S_n} \). Since these two paths terminate at the PHB of \( A_i \), they must intersect at some broker \( O_i \). Moreover, the paths are identical from \( O_i \) to \( A_i \).

(3) Every \( O_i \) must exist in the path from the SHB of \( S_r \) to the SHB of \( S_n \). If this were not the case, then three paths would exist: SHB of \( S_r \) to \( O_i \), SHB of \( S_n \) to \( O_i \), and SHB of \( S_r \) to SHB of \( S_n \). These paths would form a cycle which is impossible in an acyclic graph.

So, for any given advertisement \( A_i \), \( S_n \) only needs to be propagated to \( O_i \). Then, \( S_r \) can help \( S_n \) to show \( S_n \)’s interest in the identical path. Since \( O_i \) must exist on the path between SHB of \( S_r \) and SHB of \( S_n \), we can conclude that if a covering subscription \( S_r \)
exists, then \( S_n \) is only propagated along a portion of the path between the SHBs of \( S_n \) and SHB of \( S_r \).

To illustrate the above property, consider the example in Fig. 4.1, where subscription \( S_r \) matches advertisements \( A_1-A_5 \), and so the propagation tree of \( S_r \) is rooted at broker \( A \) with leaves at brokers \( I, J, K, L, \) and \( M \). \( S_n \) is covered by \( S_r \), and \( S_n \) only matches \( A_1, A_3, \) and \( A_5 \). From the figure we can see that the PHB of \( A_1 \) is broker \( I \), and broker \( B \) is the intersection of paths \( A-I \) and \( E-I \), so \( O_1 \) is \( B \), and similarly, \( O_3 \) of advertisement \( A_3 \) is \( C \), \( O_5 \) of advertisement \( A_5 \) is \( D \), so the subscription \( S_n \) only needs to be propagated from broker \( A \) to broker \( D \).

There are three cases that arise from Property 2 when we have a subscription \( S_n \) covered by \( S_r \): (1) The furthest \( O_i \) is \( S_n \). In this case, the subscription path of \( S_n \) is the single node consisting of the SHB of \( S_n \). (2) The furthest \( O_i \) is \( S_r \). Here the subscription path of \( S_n \) is from \( S_n \) to \( S_r \). (3) The furthest \( O_i \) is some broker between \( S_r \) and \( S_n \). The subscription path of \( S_n \) is then from \( S_n \) to the furthest \( O_i \).

Intuitively, a non-root subscription \( S_n \) only needs to propagate far enough to “graft” onto an existing dissemination tree constructed by a covering subscription \( S_r \), thereby avoiding the remaining propagation cost.

### 4.1.2 Subscription similarity

When a client removes a root subscription \( S \), the subscriptions quenched by \( S \) will be activated. We denote these subscriptions as \( NRS(S) \), the new root subscriptions of \( S \). By Property 2, the propagation of each subscriptions in \( NRS(S) \) formed a path before \( S \) was removed, but may become a tree after \( S \) is removed. Since there may be many subscriptions in \( NRS(S) \), the operation triggered by the removal of \( S \) can be very expensive and cause congestion.

However, at specific brokers, there may be little benefit in removing the old filter of \( S \) and propagating new filters for \( NRS(S) \). We need to develop a metric to find out those
brokers by quantify the similarity between $S$ and $NRS(S)$, once the filter functions of $S$ and $NRS(S)$ are in proximity, some portions of the subscription propagation tree $T_S$ can be incrementally preserved as an imprecise aggregation of $NRS(S)$.

It is difficult to quantify similarity using only the subscription filters. First, determining the filter similarity between $S$ and $NRS(S)$ involves computing a merger of $NRS(S)$, which is NP-hard [24].

Moreover, the subscriptions may not be representative of the distribution of publications received by the subscribers. Suppose a root subscription $S_1$ indicates an interest for values in the range $[0, 100]$, and a covered subscription $S_2$ has an interest in values $[0, 50]$. Based only on these filters, the best one can assume is that $S_2$ will receive half as many publications as $S_1$. However, it may be that all publications have values in the range $[0, 50]$, and so $S_1$ and $S_2$ are practically identical in terms of how many publications they actually filter.

![Diagram](image)

Figure 4.2: Relationships among $P(S)$, $P(NRS(S))$, $P(INC(S))$, and $P^*(S)$

In addition, other subscriptions that also intersect $S$ should affect our decision. For example, there may be another subscription $S_3$ with interest in the range $[40, 110]$. Notice that the range of interest of $S_3$ intersects that of $S_1$ but $S_3$ has no covering relationship with $S_1$. When $S_1$ is removed, there is no longer any filtering benefit in replacing $S_1$ with $S_2$, because $S_3 \cup S_1$ equals $S_3 \cup S_2$. In other words, those publications that are not of interest to $S_2$ need to be propagated anyway to be delivered to $S_3$.

We develop a metric that calculates similarity based on the actual history of publications...
tions that match the subscriptions in this section. First, we define some notation that will be used. For a given subscription $S$, let $INC(S)$ refer to those subscriptions that intersect $S$ but do not cover it or are covered by it, let $P(\cdot)$ denote the set of publications matched by one or more subscriptions, and let $P^*(S) = P(NRS(S)) \cup (P(S) \cap P(INC(S)))$.

$P^*(S)$ are the publications either matching $NRS(S)$ or $INC(S)$, after removal of $S$. From Fig. 4.2, we see $P^*(S)$ is a subset of $P(S)$. The similarity between $S$ and $NRS(S)$ is then computed as $\phi = \frac{\text{sizeof}(P^*(S))}{\text{sizeof}(P(S))}$, where the value of $\phi$ is in the range $[0, 1]$.

### 4.1.3 Non-decreasing similarity

Note that the similarity, $\phi$, between $S$ and $NRS(S)$ differs at every broker, and it is too expensive to record the history of publications and compute the similarity at each broker. In this section, we develop an important property. By this property, we only need to record the history of publications at the SHB of $S$, and other brokers can make use of the result computed at the SHB of $S$.

Recall that for a given publisher, a root subscription $S$ that matches an advertisement $A$ must be propagated along a path from the SHB of $S$ to the PHB of $A$. It turns out that along this path, the similarity metric defined in Sec. 4.1.2 is non-decreasing. This property which is more formally defined below, is exploited in Sec. 4.3 to incrementally prune portions of a subscription’s propagation tree.

**Property 3.** For root subscription $S$ and a matched advertisement $A$, the similarity, $\phi$, is non-decreasing at each broker on the path from the SHB of $S$ to the PHB of $A$.

**Proof.** Let $P(\cdot)$ denote the publications published by $A$, then $P^*(S) = P(NRS(S)) \cup (P(S) \cap P(INC(S)))$.

Each broker $B$ along the path from the SHB of $S$ to the PHB of $A$ may lie on the propagation path or tree of a subscription $S'$, which matches $A$ and belongs to $NRS(S)$ or $INC(S)$. Furthermore, once $S'$ is propagated to $B$, it is propagated to every subsequent
broker on the path to the PHB of $A$. This is because in an acyclic overlay, there is only one path over which $S$ and $S'$ can traverse from $B$ to the PHB of $A$.

Therefore, the subscription sets $NRS(S)$ or $INC(S)$ at a broker in the path is always a superset of those at the previous broker. This directly implies that $P(NRS(S))$ and $P(INC(S))$ are both non-decreasing, as $P(S)$ is fixed, we draw the conclusion that for a matching advertisement $A$, $\phi$ is non-decreasing.

\[ \square \]

## 4.2 Subscription packing

In this section, we develop the subscription packing algorithm. The algorithm is based on the observation that the subscriptions triggered by an unsubscription are temporally and spatially related: the temporal relationship here is simply that they are forwarded at the same time; the interests expressed by the subscriptions are also related, namely, they are covered by the unsubscription. The subscription packing algorithm exploits both these relationships by not only sending all the triggered subscriptions in a single, albeit large, message, but also by optimizing the processing of these messages using the knowledge of their covering relationships.

Before describing the packing algorithm, we first discuss some of the basic issues related to covering in publish/subscribe systems. In particular, we define three properties that any covering algorithm should satisfy, and present a simplified covering algorithm.

### 4.2.1 Traditional subscription relation graph

In many publish/subscribe systems [17, 52], a tree-like structure (as shown in Fig. 4.3) is used to maintain covering relationships and store covered subscriptions. We call this structure a subscription relation graph. The subscription relation graph is a forest of DAGs (directed acyclic graphs) where an $s_1 \rightarrow s_2$ edge means $s_1$ covers $s_2$, and there is no
subscription $s_3$ such that $s_1$ covers $s_3$ and $s_3$ covers $s_2$. Consistent with our terminology, subscriptions with no parents are referred to as root subscriptions.

![Figure 4.3: An example of a subscription relation graph](image)

When a broker receives a new subscription $S$, it must compare $S$ with all the root subscriptions. If $S$ is covered by any root subscription, $S$ is inserted as a descendant in the subscription relation graph, and need not be processed any further. Otherwise, $S$ should be immediately sent to the next stage and inserted in the graph as a new root subscription.

We define the following invariants of any subscription relation graph that both existing algorithms and our proposed algorithm must satisfy. In the properties, $S_r$ denotes the set of root subscriptions in the subscription relation graph.

**Property 4.** For all subscriptions $S \in S_r$, there does not exist another subscription $S' \in S_r$ such that $S \supseteq S'$.

Property 4 requires that root subscriptions have no covering relationships with one other. Since every new subscription has to be compared with all root subscriptions, the more the set of root subscriptions can be compacted, the less time required to determine whether a subscription needs to processed further.

**Property 5.** Suppose a root subscription $S \in S_r$ is removed. Let $S_c$ denote the children of $S$, and $S_n$ the new root subscriptions. It must be that $S_n \subseteq S_c$. 

Property 5 requires that once a root subscription $S$ is canceled, the new roots must be among the children of $S$. This property is useful during unsubscribe operations; when a root subscription is removed, the broker does not need to traverse the entire tree to find the new root subscriptions.

**Property 6.** Let $S_b$ denote the siblings of a subscription $S$. It must be that $\not\exists S' \in S_b$ where $S' \supseteq S$.

By property 6, all the children of a subscription have no covering relationships among one another. It guarantees that every node in the subscription relation graph has a minimum set of parents and children. When deleting a subscription $S$ in the graph, all the children of $S$ need to be rearranged, and property 6 ensures that only a minimum set of other nodes are affected.

The above three properties are sufficient for a broker to determine whether a subscription needs to be processed. Upon receiving a subscription, it only needs to be compared with the root subscriptions, and for an unsubscription, deleting a subscription from the subscription relation graph only involves rearranging its children. However, the subscription relation graphs in SIENA and PADRES are inefficient when it comes to inserting new subscriptions because all the covering relationships among subscriptions need to be detected. In both SIENA and PADRES, when inserting a subscription $S$ into the subscription relation graph, $S$ needs to traverse the graph to find all its parents, then $S$ needs to be compared with all the descendants of its parents to find its children. That means it needs to scan almost the entire subscription relation graph when inserting even one subscription.

### 4.2.2 Simplified subscription relation graph

As pointed out earlier, in SIENA and PADRES, the complete set of covering relationships among all subscriptions is recorded in the subscription relation graph.
Chapter 4. Incremental Filter Aggregation

The basic requirement of the subscription relation graph is to compute the root subscriptions which are the ones that need to be considered for further processing; the descendant subscriptions are quenched by the root subscriptions. In effect, descendants in the subscription relation graph are dormant, and will not be used unless they become a root subscription. In the subscription relation graph, every edge in the denotes a covering relationship. It is expensive to build all these edges, and the only use of those edges is to distinguish root subscription from descendants. Therefore it is not necessary to build the relationships among the descendant nodes. This insight forms the basis for the simplified algorithm.

When we build the subscription relation graph, we do not construct some of the edges that are expensive to find. Fig. 4.4 shows the structure of an example of the simplified subscription relation graph. It is also a forest of DAGs, and has a similar structure with the traditional subscription relation graph but with fewer edges.

![Figure 4.4: An example of a simplified subscription relation graph](image)

When inserting a subscription $S$ to the Simplified subscription relation graph, the recursive algorithm stops when the first parent of $S$, denoted $P$, is found. If any child of $P$ is covered by $S$, this child will be disconnected from $P$ and become a child of $S$. This is detailed in Algorithm 1.
Algorithm 1: Simplified subscription relation graph: recursive insert

```
Input: $S \leftarrow$ a subscription message, $P \leftarrow$ the virtual node
1 if $P$ has no children then
2    connect($P$, $S$);
3    return;
4 end
5 forall the item $C$ in $p$.children do
6    if $C$ is visited then
7        continue;
8    end
9 else
10       relation $\leftarrow$ getRelation($S$, $C$);
11       if relation = SUBSET then
12          recursiveInsert($S$, $C$);
13          return;
14       end
15       else if relation = SUPERSET then
16          direct_children_list2.add($C$);
17       end
18       else if relation = EQUAL then
19          combine($S$, $C$);
20          return;
21       end
22 end
23 connect($P$, $S$);
24 forall the item $C$ in direct_children_list do
25    connect($S$, $C$);
26    disconnect($P$, $C$);
27 end
```

4.2.3 Subscription packing algorithm

We are now ready to outline the subscription packing algorithm. The core idea is that an unsubscription and the set of associated $NRS(S)$ are forwarded in one message. In particular, all the subscriptions in $NRS(S)$ are sent as a payload of the unsubscription of $S$.

Each message in $NRS(S)$ needs to be inserted into the matching engine so they can be forwarded to the last hops of their matching advertisements. By the covering relationships among these messages, however, we know that the advertisements matching $S$ must be a superset of those matching the subscriptions in $NRS(S)$, and therefore the subscriptions in $NRS(S)$ need to be forwarded to only a subset of the neighbors where
Chapter 4. Incremental Filter Aggregation

S was forwarded. Therefore, we only need to match \( NRS(S) \) against the advisements that match S, thereby reducing processing time.

When removing S from and inserting \( NRS(S) \) to the subscription relation graph, we can combine all the deletion and insertion operations into three steps. The first step is a recursive insert of all the \( NRS(S) \) from S. Then we can remove S from the subscription relation graph and flush the \( NRS(S) \). Finally, we check if any direct child of S, denoted as C, is covered by another subscription \( S' \). If so we recursively insert C from \( S' \), and otherwise we insert C into \( NRS(S) \) and deliver it to the next broker.

This algorithm not only reduces the number of messages sent, but also reduces the total cost of subscription covering and matching. However, we see in evaluations in Sec. 4.4 that although the packing algorithm can decrease some of processing overhead, it can still result in congestion.

4.3 Incremental Subscription Tree Pruning

As pointed out in Sec. 4.1, the similarity between S and \( NRS(S) \) increases along path of \( T_S \). For every advertiser A, there exists a critical broker between the SHB of S and the PHB of A, where the similarity \( \phi \) is large enough that it is more beneficial to replace S with \( NRS(S) \) before this broker, and preserve S after this broker. Our pruning method seeks to determine the critical brokers in a distributed manner with little overhead. Since our algorithm is based on the actual history of publications, every SHB of subscription S records information about the publications matched by S. A point worth emphasizing is that only the SHB of S needs to maintain statistics on a window of publications that match S. Notably, a broker does not record the publications that only traverse through it. The statistics at the SHB are used by a lightweight and fully distributed algorithm to incrementally prune portions of a subscription’s propagation tree.
4.3.1 Statistics collection

As we mentioned above, only the SHB of $S$ needs to record some statistics on a window of publications for $S$. To collect the necessary statistics, two values are added to the header of publications, one is $distance$, which records the number of overlay hops between the SHB of a subscription $S$ and the nearest broker with another subscription $S'$ that also matches the publication. Another one is $count$, which records the number of subscriptions interested in the publication as it propagates.

At every PHB, the broker initializes the $distance$ and $count$ values of a publication to zero. At every other broker that forwards the publication, if the publication matches more than one subscription, $count$ is incremented by one and $distance$ reset to zero. Otherwise, $count$ is left unchanged, and $distance$ is incremented by one.

![Figure 4.5: Example publication propagation](image)

For the scenario in Fig. 4.5, Broker $A$ receives a publication from a publisher, and inserts the initial $distance$ and $count$ fields to the publication’s header. At broker $B$ $distance$ is incremented to one, and $count$ remains zero. At Broker $C$, the publication matches multiple subscriptions and so $distance$ is reset to zero, and $count$ incremented to one before forwarding to Brokers $D$ and $J$. This process is repeated, When the publication finally arrives at Broker $I$, both $distance$ and $count$ are two. From this,
Broker $I$ deduces that the nearest broker with a subscription that is also interested in the publication is two hops away.

<table>
<thead>
<tr>
<th>pubID</th>
<th>advID</th>
<th>subID</th>
<th>distance</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>A1</td>
<td>S1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>N2</td>
<td>A1</td>
<td>S1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>N3</td>
<td>A1</td>
<td>S1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>N4</td>
<td>A1</td>
<td>S1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>N5</td>
<td>A1</td>
<td>S1</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Portion of the database at Broker $I$ in Fig. 4.5

Each broker that is the SHB for a subscription that matches a publication records the following information: pubID, the publication identifier; advID, the advertisement identifier; subID, the subscription identifier; and the distance and count values in the publication. For example, Table 4.1 shows what the database at Broker $I$ might look like. The entries in the database can expire after a configurable time so that it maintains information about a sliding window of matching publications. Furthermore, these entries are purged when the associated subscriptions are unsubscribed.

### 4.3.2 Subscription tree pruning

Based on a given threshold of similarity, for each advertiser, the SHB can efficiently and locally determine the distance to the critical broker using the distance value. For example if the threshold of similarity is 0.85, then for advertisement $A$, if the distance value of more than 85 percent of the publications from $A$ is less than 7, then the critical broker of $A$ is 7 hops away in the path from SHB to the PHB of $A$. Then, a list of tuples is generated for the unsubscription message with each tuple containing 2 values: the identifiers of matched advertisers (advID), and the pruning distance from the SHB to the critical broker (pruneDis). These tuples are added to the header of the unsubscription message, and our pruning algorithm is performed with the propagation of the unsubscription message.
Algorithm 2: Handler of Unsubscription Message

Input: $us \leftarrow$ an unsubscription message

1. forall the neighbor $U$ of current broker do
2. generate a new tuplelist $newlist$;
3. $newlist \leftarrow \phi$;
4. forall the tuples $\{advID, pruneDis\} \in unsub.tuplelist$ do
5. if $advID.sender = U$ and $pruneDis > 0$ then
6. $newlist \leftarrow newlist \cup \{advID, pruneDis - 1\}$;
7. end
8. end
9. if $newlist \neq \phi$ then
10. $nus \leftarrow us$;
11. $nus.tuplelist \leftarrow newlist$;
12. sent $nus$ to $U$;
13. end
14. end

Each broker uses Algorithm 2 to determine whether to continue forwarding the unsubscription and prune the next hop of the subscription tree. For every neighbor $U$ of the current broker, if all the advertisements from $U$ have reached its critical broker, then it is unnecessary to delivery the unsubscription message to $U$. In other words, the remainder of the subscription tree $T_S$ is preserved.

Fig. 4.6 depicts an example of incrementally pruning a subscription propagation tree $T_S$. The white brokers indicate where $S$ has been removed and replaced with subscriptions in $NRS(S)$, the dark brokers are where $S$ has been preserved.

4.4 Evaluation

In this section, we experimentally compare our incremental unsubscription algorithm with traditional covering optimizations. The objective is to evaluate the performance of our algorithm under a variety of workloads and system load characteristics.
4.4.1 Methodology

Default setup

The protocols described in this chapter have been implemented in the Java-based PADRES content-based pub/sub prototype. The experiments are run on a cluster of 21 machines each with four 1.86 GHz Xeon processors and 4 GB of RAM. This setup mimics a data center environment and offers a controlled system from which we can derive meaningful analysis.

The network topology in the experiments consists of 49 brokers, as shown in Fig. 4.7, the 5 brokers labeled $A$–$E$ are denoted as core brokers. Publishers connect to the 3 edge brokers close to broker $E$, and subscribers are randomly distributed among the remaining edge brokers. This might represent a messaging platform for the business workflows in a large enterprise with multiple departments and partners. The components in each workflow would subscribe to triggers of interest, and these workflows are invoked by publications from external clients [57, 103, 67].

The degree of covering among the subscriptions is quantified by the covering degree which is the average number of new root subscriptions triggered by an unsubscribe oper-

---

1Available at [http://padres.msrg.toronto.edu](http://padres.msrg.toronto.edu).
ation. Let $R_i$ denote the number of subscriptions in the set $NRS(S_i)$, and $N$ represent the number of root subscriptions. Given a set of subscriptions, the covering degree is then defined as $CovDeg = \frac{\sum_{i=1}^{N} R_i}{N}$. The default subscription covering degree in the experiments is 2100.

**Algorithms**

The experiments compare the incremental algorithm proposed in this chapter with three other algorithms described below.

**Active covering:** Active covering is the traditional covering algorithm. When a broker encounters a new subscription $S$ that covers existing uncovered subscriptions $S$ from the same neighbour, it forwards $S$, then forwards unsubscriptions for $S$ and then deletes $S$ from its routing table to keep the routing tables compact. Similarly, when $S$ is unsubscribed, it must again forward all the subscriptions $S$ to ensure publications are routed correctly.

**Lazy covering:** In the lazy covering algorithm, when a new subscription $S$ covers existing uncovered subscriptions $S$, the broker simply forwards $S$, and when $S$ needs to
unsubscribed, only subscriptions after $S$ may need to be forwarded. While this algorithm forwards fewer subscriptions, over time it results in larger routing tables and hence slower publication matching and delivery performance.

Subscription packing: Subscription packing is an optimization of the active covering algorithm: when a set of subscriptions or unsubscriptions $S$ need to be forwarded at once, they are forwarded in one message. This algorithm not only reduces the number of messages sent, but we utilize the knowledge that these messages have a covering relationship to more efficiently index them in the covering data structure. Note, however, that there is no benefit in matching and forwarding these subscriptions as they must be processed individually.

Metrics

The experiments measure a number of metrics that are useful in understanding how the algorithms perform.

Publication propagation delay: This is the end to end latency of publications as they traverse from a publisher to a subscriber.

Input queue size: This is the number of messages waiting to be processed by a broker. In the experiments, the queue size is used as a measure of congestion in the network, and shows how quickly the system stabilizes after a set of unsubscribe operations.

Routing table size: This represents the number of subscriptions stored in the routing table of a broker and also reflects the amount of subscription messages transmitted among brokers. Smaller routing tables consume less memory and result in faster routing computations.

Triggered NRS messages: This metric measures the number of hops traversed by the NRS subscription messages that are triggered by an unsubscribe operation.

False positive publications: This is a count of the number of publications that are unnecessarily forwarded towards uninterested subscribers. Only the incremental algorithm
has false positives.

### 4.4.2 Effect of covering

Before studying how unsubscriptions can induce congestion, we first present the benefits of the covering algorithms in the absence of unsubscriptions. In this experiment, we issue 42,000 subscriptions and 200,000 publications and measure how long brokers spend processing subscriptions (split into the covering detection and matching phases), unsubscriptions, and publications. In Table 4.2, we see that the active covering algorithm expends more cycles detecting covering relationships, but can process publications in half the time. Therefore, active covering is a good choice when the subscription set is relatively stable and publication delivery latency is important. As there are no unsubscriptions issued in this workload, we do not see the subscription processing benefits of lazy covering here, but we do note that its publication processing delay is larger than with active covering, and will only get worse with subscription churn.

We further emphasize that although subscription covering can reduce the publication matching time, the cost of maintaining the covering relationships among subscriptions is also expensive [94]. For example, Table 4.2 shows that this covering computation is an order of magnitude more expensive than subscription matching and dominates the subscription processing pipeline. Since an unsubscription can trigger many subscriptions, the covering computation of those triggered subscriptions is the main cause of the system

<table>
<thead>
<tr>
<th></th>
<th>Active</th>
<th>Lazy</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub matching</td>
<td>14.5 (0.81)</td>
<td>28.8 (1.6)</td>
<td>33.7 (1.6)</td>
</tr>
<tr>
<td>Sub covering</td>
<td>216 (12)</td>
<td>486 (27)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>Unsub</td>
<td>11.7 (1.4)</td>
<td>0 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>Pub</td>
<td>600 (3.0)</td>
<td>1240 (6.2)</td>
<td>1580 (7.9)</td>
</tr>
<tr>
<td>Total</td>
<td>842</td>
<td>1754</td>
<td>1613.7</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of total broker processing time (in seconds) for different message types under active covering, lazy covering, and no covering algorithms. (The average processing time per message (in milliseconds) is shown in parentheses.)
The remainder of the experiments will show how the incremental algorithm can achieve the low publication processing times of the active covering algorithm without the associated unsubscription induced congestion.

### 4.4.3 Comparisons among algorithms

The first set of experiments compare the proposed incremental algorithm with the traditional active covering algorithm, as well as the lazy covering, and packing algorithms. In this group of experiments, 80,000 subscriptions with covering degree 2100 are issued before we trigger unsubscribe operation. To make our experiments more realistic, we issue a continuous stream of background publications at a rate of one every 10 ms. The similarity threshold for the incremental algorithm is set at 90%, which implies a 10% publication false positive rate. Then, a set of 20 unsubscriptions are sent, one every 0.5 s. After the unsubscription operations, the experiment is allowed to run for several hours to measure how the algorithms process any resulting message bursts and congestion.

First, we examine the propagation delay of publications. Fig. 4.8(a) plots how the propagation delay for the four algorithms vary over time. The active covering algorithm suffers from peak delays of over 2400 s with average delays of over 1000 s during the experiment. The lazy covering and subscription packing algorithms deliver publications in about 60% of this time on average. The incremental unsubscription algorithm, however, performs substantially better, with average delays of only about 0.4 s, an improvement of more than 99%. Moreover, congestion subsides quickly with the incremental algorithm, and the average steady state publication delay falls to 200 ms. Next, we will investigate the causes behind these publication delays.

Fig. 4.8(c) plots (on a log scale) the number of triggered \(NRS\) messages at the core brokers \(A-E\), and shows that with active and lazy covering, as well as subscription packing, the number of \(NRS\) messages increases rapidly at brokers approaching the
Chapter 4. Incremental filter aggregation

Figure 4.8: Comparisons of active covering, lazy covering, subscription packing, and incremental unsubscription algorithms

Publisher, which in this case is broker $E$. For example, with active covering, there are about 700 messages triggered at broker $A$ increasing to almost 20,000 messages at broker $E$. The incremental algorithm, by selectively stopped subscription dissemination at a critical broker, only propagates about 1100 $NRS$ messages at broker $E$. Fig. 4.8(c) also confirms the claim made in Sec. 4.1.3 that the incremental algorithm is more inclined to preserve subscription trees at brokers closer to a publisher. Finally, the results show that the number of triggered $NRS$ messages at the remaining inner brokers is much less than that at the core brokers.

As most of the congestion occurs at the core brokers, we now focus on how the input queue of these brokers are affected by the messages triggered by the unsubscriptions. As shown in Fig. 4.8(b), the queue size of the core brokers explodes to almost 24,000 mess-
sages with the active covering algorithm. The queue size in the lazy covering and packing algorithms are smaller but still experience a large burst of up to 1500 messages. However, the incremental algorithm presented in this chapter far outperforms all of these with only a slight increase in the input queue size peaking at 150 messages. More importantly, the incremental algorithm stabilizes very quickly, while the active and lazy covering only manage to slightly reduce the queue size even after one hour into the experiments. These large long-lived queues indicate that the system is overloaded and cannot function normally. Seeing how the delays in Fig. 4.8(a) track the queue size in 4.8(b), it is evident that the delays are a result of the congested queues.

Furthermore, the large publication delays in Fig. 4.8(a) are consistent with queuing theory results which show that queuing delays increase rapidly as the message arrival rates approach the link capacity [48].

It should also be noted that the burst of subscriptions in $NRS(S)$ triggered by an unsubscription of $S$ do not only occur when the unsubscription is first issued. Instead, as the unsubscription makes its way through the network, it may trigger, at each broker along the unsubscription’s propagation tree, the immediate forwarding of the subset of $NRS(S)$ that exists at that broker. This explains why the queue lengths in Fig. 4.8(b) fluctuate; as an unsubscription propagates slowly through the network, it will sometimes arrive at a broker with many covered subscriptions and cause these subscriptions to be forwarded thereby giving rise to another subscription burst. Moreover, these triggered subscriptions themselves may affect the covering relationships at the downstream brokers, causing perhaps even more congestion.

The subscription packing algorithm decreases the number of triggered $NRS$ messages by including them within a single message envelope, and tries to index all these messages as a whole in the covering data structure. Compared to the incremental algorithm, however, the benefits of packing are limited. Fig. 4.8(b) demonstrates that the message processing cost dominates network overhead and the network can still be severely con-
gested with thousands of publications and subscriptions pending in the brokers’ queues.

In addition to triggering fewer messages, the incremental algorithm also resulted in smaller routing table sizes in this experiment. Fig. 4.8(d) plots the average routing table size at the core brokers at the end of the experiment after the unsubscriptions have been issued. We see that while the active covering algorithm achieves compact routing tables. The lazy covering algorithm, on the other hand, relies on the order that subscriptions are issued and only opportunistically benefits from covering relationships and thus results in larger routing tables. However, the incremental algorithm most aggressively aggregates subscriptions and achieves smaller routing table size for it gives more opportunities for filter aggregation. It should be noted that the routing table size benefits of the incremental algorithm only manifest when covering subscriptions are unsubscribed, so it may not always outperform the active covering algorithm.

One overhead of the incremental algorithm is the gathering of statistics on a window of publications. Recall, however, that only the subscriber hosting brokers are required to maintain this information. In this experiment this data was stored in an Apache Derby database, the largest of which contained about 28,000 entries requiring about 24 MB of storage. The average broker’s database was an even more modest 12 MB. More importantly, the results presented above on a real running implementation of the protocol show that, in spite of any overhead, the incremental algorithm far outperforms the traditional ones.

4.4.4 Varying number of unsubscriptions

In this section, we investigate the effects of varying the number of unsubscriptions issued. Since the results in Sec. 4.4.3 showed that the performance of the active covering, lazy covering and packing algorithms are similar, here we compare the incremental algorithm with only the active covering algorithm—the most widely used one.

Fig. 4.9 shows the performance of the algorithms when the number of unsubscriptions
issued over a 10 s interval varies from 20 to 60.

The publication delay results in Fig. 4.9(a) show that the congestion induced by just 20 unsubscription messages causes the active covering algorithm’s messages delivery to take over 260 s on average. Meanwhile, the incremental unsubscription algorithm maintains a stable and fast delay of less than half a second.

The cause of the publication delays is the congestion arising from the subscription bursts in the active covering algorithm. Figs. 4.9(d), 4.9(e), and 4.9(f) show how the magnitude and duration of the input queues increase with the number of unsubscriptions under the active covering algorithm. The incremental algorithm, however, is much better at avoiding large congestion.

### 4.4.5 Varying subscription covering degree

This section investigates the effects of varying the covering degree among the subscriptions in the workload from a low degree of covering of 700 to a high of 2100.

Beginning again with the publication delays, Figs. 4.10(a), 4.10(b), and 4.10(c) show
that the active covering algorithm is sensitive to the degree of covering in the subscrip-
tion population but the incremental algorithm provides relatively stable performance.
Moreover, the results indicate a wide distribution in the publication delays under the
active covering algorithm.

Our results show that after unsubscribe operations, the routing table sizes sharply
increase with increasing covering degree of subscription workloads: from just under 8000
to over 21 000 for the active covering algorithm. More interestingly, since a larger covering
degree gives more opportunities for filter aggregation, the incremental routing table sizes
decrease with increasing covering degree of subscription workloads. All in all, the benefits
of the incremental algorithm accrue as the covering degree increases.

Figs. 4.10(d), 4.10(e), and 4.10(f), show that workloads that exhibit more covering
relationships among subscriptions are more likely to congest the system resulting in large
queue lengths. This is understandable since more covering relationships are positively
correlated with larger $\textit{NRS}(S)$ sets, which are prone to lead to subscription bursts when
the root subscription $S$ is removed. For example, when the covering degree is 10, the
active covering and incremental algorithm both result in relatively minor and short-lived congestion. When the covering degree increases to 1400, the active covering algorithm, in Fig. 4.10(e), experiences bursty traffic with queue lengths of about 10 000 messages. This trend continues with the active covering performing worse with larger covering degrees; when the covering degree is 2100, the queue sizes continue to increase an hour into the experiment and remain even after two hours. Contrast this behavior to the incremental algorithm, in Fig. 4.10(f), which only experiences a short-lived congestion of less than 150 messages for all covering degrees.

4.4.6 Varying similarity threshold $\theta$

We now focus on the incremental algorithm and investigate the effect of the similarity threshold $\theta$. Recall from Sec. 4.1.2 that a smaller $\theta$ indicates that the algorithm should be less aggressive in removing subscriptions, thereby accepting more false positive publications but triggering fewer congestion inducing subscriptions.

Table 4.3 summarizes the routing table size and publication delay results for the case with 20 unsubscriptions. We see that, indeed, a lower $\theta$ will trigger fewer subscriptions. With a very low threshold of $\theta = 0.1$, we observe that no unsubscription or subscription messages are forwarded. The consequence of this is that there is less congestion and publication delivery delays are kept low. The tradeoff is that more false positive publications are attracted.
4.4.7 Distributed workflow scenario

In this experiment we evaluate a distributed business process execution engine built over a publish/subscribe overlay. The broker topology consists of four domains, each representing a business partner, and each domain contains ten brokers arranged in a three level hierarchy. Within each domain are six process execution engines that can host and execute activities within a process [57].

Three classes of business processes are deployed across these execution engines. A small process contains three activities all of which are deployed to a single randomly chosen engine; a medium process contains ten activities which are deployed across the engines within a randomly chosen domain; and a large process has 26 activities each of which are deployed to a random engine in the system. The latter represent processes that involve multiple business partners. There are 80 small, 80 medium, and 40 large processes for a total of 200 processes. These processes contain combinations of parallel and sequential control flows, as well as loops [57]. Once deployed, the processes are periodically triggered and the associated engines issue advertisements, subscriptions, and publications to coordinate the flow of each process instance.

In addition, 30% of the processes are monitored by a client that subscribes to the associated control flow messages. This could represent a developer monitoring a process, or a business analyst tracking certain metrics about the process. Furthermore, four clients periodically subscribe to all process trigger messages for a period of one minute every ten minutes. This may be useful to an administrator sampling the system in order to profile it, or an auditor randomly inspecting certain process behaviour.

Fig. 4.11(a) shows periodic spikes in publication delivery delays with active covering that corresponds to when a monitoring client stops monitoring and issues an unsubscript. The incremental algorithm, however, maintains stable delays, averaging about 0.24 s.

Turning our attention now to the queue size in Fig. 4.11(b), we see the cause of
Chapter 4. Incremental filter aggregation

Figure 4.11: Distributed business process scenario

the above delays is due to congestion triggered by the periodic unsubscription messages. Again, however, the incremental algorithm suffers virtually no congestion.

Furthermore, Fig. 4.11(c) shows how the routing tables experience large fluctuations as the active covering algorithm repeatedly attempts to optimize the entries in the table. The incremental algorithm, also reacts to changing subscriptions but the modifications to the routing tables are much less severe.

4.4.8 Complex publish/subscribe matching engine

The remaining set of experiments evaluate the algorithms with an earlier unoptimized version of the PADRES system. In particular, it used a simple subscription covering detection structure with similar performance to that in Siena [16]. The results obtained can be representative of a more expressive publish/subscribe data model, such as XML data and XPath queries, that incurs more processing overhead.

Fig. 4.12 compares the algorithms with the complex publish/subscribe matching engine. The experiment begins with a subscription phase in which 30,000 are issued with an interval of 1500 ms between subscriptions. Then 300 of these subscriptions are unsubscribed at a rate of one every 500 ms. As well, slightly over 21,000 publications are issued over the experiment at the rate of one every 400 ms. While publications continue throughout the experiment, the delay results only consider a set of 300 publications matching subscribers at all 32 edge brokers that are issued 600 s after the unsubscribe
operations.

Fig. 4.12(a) plots the average propagation delay for the four algorithms, along with 10 and 90 percentile values in the distribution of delays. The results show that the average publication delay across the experiment with the active covering algorithm is over 4400 s, while the lazy covering and subscription packing algorithms deliver publications in less than half this time. As in earlier experiments, the incremental algorithm’s average delays are only about 0.4 s. Moreover, congestion subsides quickly with the incremental algorithm, and the average steady state publication delay falls to 200 ms. The results in Fig. 4.12 are largely consistent with the earlier results with a faster matching engine. The main difference is that we observe even more pronounced congestion that persisted for almost three hours with the active covering algorithm. The results indicate that the vulnerability of the system to unsubscription triggered bursts is sensitive to the broker processing time.

Fig. 4.13 shows the performance of the algorithms when the number of unsubscriptions issued over a 150 s interval varies from 100 to 300. The publication delay results in Fig. 4.13(a) shows that both the active covering and incremental algorithms perform well, delivering publications in about 20 ms when there are only 100 unsubscriptions. However, the performance of the active covering algorithm quickly and substantially degrades to over 1200 s and 2000 s as the number of unsubscriptions increases to 100 and 300, respectively. Meanwhile, the incremental algorithm maintains the same publication delay performance. Compared to earlier results, we see that the duration of congestion is much longer, as much as three hours.

In Fig. 4.14, we show the effects of varying the covering degree among the subscriptions in the workload from a low degree of covering of 10 to a high of 50. The conclusions are the same as in the earlier experiments, but congestion is much more severe with the complex matching engine; with the active covering algorithm and a covering degree of 50, we observe queues are still congested nine hours into the experiment.
Figure 4.12: Comparisons of active covering, lazy covering, packing and incremental algorithms (complex matching engine)

Figure 4.13: Varying number of unsubscriptions (complex matching engine)

Figure 4.14: Varying subscription covering degree (complex matching engine)
4.4.9 Clustered placement

We now evaluate the algorithms using a workload where the subscriptions are clustered in the network [85]. In particular, subscriptions with covering relationships are distributed to the same cluster, with each cluster consisting of 2 to 4 adjacent brokers as depicted by the regions in Fig. 4.7. We revert to the default subscription covering degree of 30 for this experiment.

Fig. 4.15(a) shows the average propagation delay of 300 publications issued 600 s after the unsubscribe operations. We see that the incremental algorithm delivers publications in about 50 ms, which is less than 3% of the 2000 s average delays experienced when the active covering algorithm is used.

In terms of the routing table sizes in Fig. 4.15(b), the incremental algorithm achieves substantially smaller tables. For example, among core brokers, the average table size with the incremental algorithm is about 1200 compared to about 7300 for the active covering algorithm, an impressive 82% reduction. What we see here is that clustering subscriptions makes it more likely that a root subscription can be left in place without suffering from too many false positives.

The input queue lengths are plotted in Fig. 4.15(c). We see that the active covering algorithm performs much worse when subscriptions are clustered since this tends to concentrate the effects of the congestion caused by the subscriptions in $NRS(S)$. Despite having issued all the unsubscriptions by 150 s into the experiment, the message bursts
only begin to drop after 4000 s. The incremental algorithm, however, has no congestion because the clustering of subscriptions in the topology offers more opportunities for filter aggregation, and almost all the triggered \( NRS \) are replaced by a broader subscription. As well, compared to Fig. 4.12(e), we observe that the benefits of the incremental optimization are more evident when the subscriptions are clustered instead of scattered.

### 4.4.10 Mobile scenario

The following set of experiments consider a more dynamic scenario in which subscriptions are unsubscribed and resubscribed at different brokers. The experiment is designed to reflect scenarios such as a distributed gaming application.

![Tiered topology for the mobile scenario](image)

The setup consists of a 5000 subscribers and 40 publishers distributed randomly across the brokers in the topology shown in Fig. 4.16. In a distributed online game, each broker may be responsible for managing a portion of the game world, so as players move in the game, they will reconnect to different brokers in the network [45, 12].

In addition, there are four subscriptions, one per region in Fig. 4.16 that are interested in all publications. We refer to these subscribers as monitors and they may represent interest management servers in an online game. As load conditions may trigger migrations of these interest management servers, these subscriptions may need to be resubscribed.
at other brokers.

We consider two cases: one where only the monitors move at a rate of one movement every 2 minutes for a total of 6 hours, and another where in addition to the movement of the monitor subscriptions, the others subscriptions move at a rate of 60 movements per minute for the entire duration of the experiment. The non-monitor subscriptions only move within their region in the topology. In both cases, publishers scattered among the brokers do not move.

Fig. 4.17(a) shows the congestion at the core broker when only the monitor subscriptions move. We see that the active covering algorithm suffers from a large and growing input queue, whereas the incremental algorithm has virtually no congestion. When all subscriptions move in Fig. 4.17(b), there is no significant change in the congestion, indicating that it is the movement of the broad monitor subscriptions that cause the congestion.

In terms of the routing tables, Fig. 4.17(c) shows that with the active covering algorithm the average number of entries in the broker routing tables fluctuates during the experiment. These fluctuations are a consequence of the active covering algorithm attempting to maintain compact routing tables when covering subscriptions are removed, particularly the monitoring subscription. This is why there the introduction of the movement of smaller subscriptions in Fig. 4.17(d) has little effect on the routing table size. The incremental algorithm, on the other hand, has a much smaller and more stable routing state in both cases. The reason for the stable performance is that the incremental algorithm effectively maintains the monitor subscriptions even after they move away, and so after a few movements, these subscriptions exist virtually everywhere in the network and “hide” the movement of the other subscriptions. Interestingly, this means that with subscription pruning, the presence of movement can actually result in smaller routing tables than when there is no movement.

Of course, preserving these subscriptions also increases the false positive publication
traffic, but it turns out that this traffic is distributed mostly among the more lightly loaded brokers in the system. Fig. 4.17(e) shows the average number of publications traversing each tier of brokers in the topology. In both algorithms, the brokers in the upper tiers are more heavily loaded, with the incremental algorithm imposing more traffic due to the false positives. However, when we isolate this false positive traffic in Fig. 4.17(f), we see that it is the more lightly loaded lower tier brokers that are required to carry most of the false positive load, helping to distribute publication load.
Figure 4.17: Mobile scenario (complex matching engine)
Chapter 5

Distributed execution engine

This chapter develops the NIÑOS distributed process execution engine. Section 5.1 describes how a BPEL process is mapped to the NIÑOS system architecture, and Section 5.2 presents an evaluation of this platform.

5.1 Distributed process execution

NIÑOS is a distributed business process execution architecture. It leverages the PADRES publish/subscribe system by transforming a BPEL business process into fine-grained publish/subscribe agents that collaborate to realize the original process. These agents interact using publish/subscribe messages and take advantage of some of the in-network processing capabilities available in PADRES. To simplify management, NIÑOS allows processes to be deployed and monitored in a centralized manner, again exploiting some of the decoupling properties of the PADRES publish/subscribe system.

5.1.1 Execution architectures

The simplest business process engine consists of a single execution engine as shown in Figure 5.1(a). This centralized engine is responsible for executing and managing all
concurrent instances of the processes deployed on it. The advantage of such an architecture is its simplicity in terms of deployment and management. However, as the resources in such an architecture are fixed, the system may not scale with the complexity of processes and the number of process instances. Also, the single execution engine represents a single point of failure and may not be appropriate for the execution of mission critical processes. Furthermore, inter-organization business processes may have no obvious choice for a central coordinator.

To address scalability and fault-tolerance, a cluster of execution engines can be deployed. In this architecture, illustrated in Figure 5.1(b), each engine in the cluster is essentially a replica of the others, and can execute a complete business process. A call to a business process $P$ is first sent to a load balancing component (not shown in the figure), which forwards the call to one of the engines $E$ in the cluster, based on some criteria such as ensuring a balance of load across the cluster. At this point, engine $E$ creates an instance of process $P$ and is responsible for executing the instance until completion. Some systems support the ability to add and remove engines to the cluster as the load varies. A clustered execution architecture can be scalable and does not suffer from a single point of failure. However, process instances are still executed in a centralized manner, and control and data is still concentrated in the cluster. Consider the case of a data-intensive business process that transfers and operates on large volumes of data. In a clustered architecture, the data needs to be transferred to the cluster before it can be operated on by the process. In a more flexible deployment it would be possible to move the portions of the process that operate on the data closer to the data source thereby

![Figure 5.1: Execution engine architectures](image-url)
reducing the time and network costs incurred in having to transfer the data.

This thesis proposes an execution engine in which processes themselves can be distributed. As shown in Figure 5.1(c), a process is first decomposed into tasks. In a BPEL process, the tasks can be the individual BPEL activities. These tasks are then assigned to various execution engines in the system. We emphasize that these execution engines can be light-weight as they only execute fine-grained tasks, as opposed to complete processes. A key benefit of such an architecture is the ability to deploy portions or processes close to the data they operate on, thereby minimizing bandwidth and latency costs of a process. For example, for data intensive business processes, such as rendering farms or large simulations, it would be possible to deploy only those portions of the process that require access to large data sets close to their respective data sources. Different parts of the process that operate on different data sets can be independently deployed near their respective data sources. This is not possible in a clustered architecture since the entire process instance must be executed by a single engine.

5.1.2 NIÑOS system architecture

The NIÑOS system architecture, as shown in Figure 5.2, consists of four components: the underlying PADRES broker network, activity agents, Web service agents, and a business process manager. As mentioned in Section 2.1, the PADRES broker network consists of
a network of brokers that carry out content-based routing and in-network processing of composite subscriptions.

In NIÑOS, each business process element, such as a BPEL activity, has a corresponding *activity agent*, which is a light-weight publish/subscribe client. Generally, an agent waits for its predecessor activities to complete by subscribing to such an event, then executes its activity, and finally triggers the successor activities by publishing a completion event. As a result, process execution is event-driven and naturally distributed.

Cross-enterprise business interaction is a requirement in business processes. For example, BPEL supports invoking partner Web services. NIÑOS *Web service agents* interface Web services with the PADRES network by translating between Web service protocols (such as SOAP over HTTP) and publish/subscribe message formats. This allows the appropriate activities in a NIÑOS business process to invoke and be invoked by external Web services. Web service agents support both static partners, which are defined at design time, and dynamic partners, determined at runtime.

The *business process manager*, which is also a publish/subscribe client, transforms business processes into publish/subscribe messages for the activity agents, deploys the process onto the available agents in the network, triggers instances of business processes, and monitors and controls the execution.

NIÑOS addresses three phases of business process execution: process transformation, deployment, and execution. In the *transformation* phase, a business process is mapped to a set of activity agents and corresponding publish/subscribe messages that specify the dependencies among the activities. The transformation of some interesting BPEL activities is described in Section 5.1.3 in detail.

In the *deployment* phase, the business process manager deploys the process to the appropriate activity agents. Each activity agent subscribes to *agent control messages* with a unique agent identifier, allowing the manager to install an activity at a particular agent. An agent partakes in a business process by issuing the subscriptions and advertisements
as requested by the manager, thereby building up the inter-agent activity dependencies and making the process ready to execute.

In the execution phase, the deployed business process can be invoked through a Web service agent, which translates the invocation into a NIÑOS service request. The service request is a publication message that specifies the process and instance identifiers, and other required information. The first activity agent in the process, say the receive activity in the process in Figure 5.2, receives this publication, instantiates a process instance, processes the activity, and triggers the successor assign activity. Agents execute and trigger one another using publish/subscribe messages in this event-driven manner until the process terminates.

Unlike a centralized orchestration engine, the NIÑOS agent-based engine supports flexible deployment. All activity agents can be deployed at one node, effectively executing processes in a centralized manner, or distributed across the network to realize fully distributed execution. It is also possible to cluster sets of agents and to achieve partially distributed execution. By automatically and dynamically deploying agents at strategic points in the network based on network conditions and available system resources, the NIÑOS execution engine can optimize the business processes. Such QoS-based business process execution is one of the ongoing research directions for this system [22, 69].

The PADRES and NIÑOS system architecture is conceptually summarized in Figure 1.1. A set of computing and network resources are virtualized by the PADRES distributed content-based publish/subscribe routing layer. Over this layer a set of distributed NIÑOS agents collaborate and coordinate to execute a business process. Finally, various tools are available to monitor and manage the process execution and the publish/subscribe layer.
5.1.3 Process transformation

NIÑOS supports the transformation of the complete set of BPEL features, including fault, compensation, and event handling. This section outlines the transformation of some of the more interesting BPEL activities from Table 2.2, notably the while, pick, compensate, switch, and flow activities.

While activity

The BPEL while activity repeatedly executes a sequence of activities until a condition, which is a Boolean expression on BPEL variables, is no longer satisfied.

A generic use of the while activity is shown in the BPEL process fragment in Figure 5.3, where the italicized activities are placeholders for one of the standard BPEL activities. The while activity is mapped to a while agent that evaluates the condition expressed in the activity, and triggers the appropriate subsequent activity. In NIÑOS, the while agent evaluates the while condition at the beginning of each iteration of the loop. In order to be triggered at this time, the while agent issues the subscriptions Sub1 and Sub2 in Figure 5.3. These subscriptions are matched by the successful completion of the activity preceding the while activity, or by the final activity within the while loop.
As well, the *while* agent issues publications Pub1 and Pub2 in Figure 5.3 to trigger another iteration of the while loop or to exit the loop and continue execution with the first activity following the loop.

Although not shown, the *while* agent mapping also specifies the subscription and publication messages for the activity preceding the *while* activity (activity1 in Figure 5.3), the first and last activities within the while loop (activity2 and activity3), and the first activity after the loop (activity4). Also not shown are the messages used to assign and retrieve variables. For example, the *while* activity may subscribe to update publications for any variables used in the while condition. The handling of BPEL variables is discussed further in Section 5.1.3.

**Pick activity**

The BPEL *pick* activity waits for one or more events to occur and conditionally executes a sequence of activities based on the event that occurred. The events a *pick* activity can wait on include messages, such as Web service invocations or asynchronous replies, and alarms, which are triggered after some time duration or deadline.

A generic use of the *pick* activity is shown in Figure 5.4. Note that many details, such as the onMessage parameters, are omitted. The *pick* activity is mapped to a *pick* agent that blocks and listens for one of the events specified in the pick activity to occur, and then triggers the appropriate subsequent activity. The execution of the *pick* activity is triggered when the preceding activity complete, which the *pick* agent listens for with subscription Sub1 in Figure 5.4. Also, the *pick* agent issues a subscription for each onMessage it listens for (Sub2 in Figure 5.4), and when a matching event occurs, it issues a publication to trigger the appropriate activity (Pub1 in Figure 5.4).

Note that no subscriptions are issued for onAlarm events since alarm deadlines or durations are evaluated internally by the *pick* agent. As with the previous activity, not all the subscription and publications messages are shown here.
Compensate activity

Compensation handlers are an application specific rollback mechanism in BPEL. The activities in a BPEL process are grouped into arbitrarily nested scopes, and each scope may define a fault handler and a compensation handler. When a fault, or exception, occurs, the scope’s fault handler is called. A compensate activity within the fault handler can call the compensation handlers for any nested scopes that have successfully executed. A compensation handler attempts to “undo” the logic within the scope. For example, the compensation for a scope whose activities ship a product to a customer may be to cancel the order if it hasn’t been delivered yet, or otherwise notify the customer that the order cannot be canceled.

A generic use of the compensate activity is shown in Figure 5.5. Here, ScopeA’s fault handler invokes the compensation handler in ScopeB. The scope agent for ScopeB subscribes to compensation events for its scope with Sub1 in Figure 5.5, and triggers the first activity in its compensation handler using Pub1 in Figure 5.5.

BPEL semantics require the compensation handler to be called with a snapshot of the variables when the scope completed. This can be achieved by retrieving these values using the PADRES historic access capability [51], or by having each scope handler cache these values upon scope completion. These cached values would be flushed when the
process instance completes. In Figure 5.5, this would be done by ScopeB’s scope agent.

Switch activity

The BPEL switch activity allows for conditional execution, whereby one of several case branches is executed based on a Boolean condition associated with each case. The cases are ordered and the first branch whose condition evaluates to true is taken. If all the cases fail, an optional otherwise branch is taken.

Figure 5.6 gives an example of a process with a switch activity. Not illustrated in the figure is the possibility for execution to transfer directly from the switch1 activity to activity4 if neither case condition is true. In NIÑOS, a switch agent is used to evaluate the case conditions in each branch of a switch activity.

A switch agent subscribes to updates from the system for any variables necessary to evaluate the case conditions, and determines which (if any) branch should be taken. By using a composite subscription (Sub1 in Figure 5.6), the switch agent receives a single notification of its predecessor activity’s completion, along with all the required variable updates in the associated process instance. After evaluating the case conditions, the switch agent triggers the appropriate branch with a publication such as Pub1 in Figure 5.6. The first activity in each branch subscribes to these trigger publications. For example, in Figure 5.6, activity2 subscribes to Sub2. Note that the case where none of the cases in a switch activity are taken is not shown.

An alternative implementation could eliminate the need for a switch agent entirely, by transferring the responsibility of determining the appropriate branch to follow to the first activities within each case branch. For example, in Figure 5.6, the agents associated with activity2 and activity3 could independently determine if they should execute. The tradeoff, however, is that these agents will have to perform redundant computations of the case conditions. Recall that the case statements are ordered and only the first true case condition is executed. Therefore, in Figure 5.6, activity3 must evaluate the condition
that case2 is true and that case1 is false. These redundant computations are unnecessary if the conditions are evaluated by a single switch agent. Furthermore, distributing the computation of the case conditions requires sending the variable updates necessary to compute these conditions to several agents.

Flow activity

The BPEL flow activity supports the execution of parallel branches. Branches in a flow typically execute concurrently, but may be synchronized by a link. A link between a source and target specifies that the target activity executes only after the source activity has completed. An activity may be the source or target of multiple links.

In addition, a source activity may set a Boolean valued transition condition on its outgoing links based on an expression of its process instance’s state. Likewise, a target activity may specify a Boolean valued join condition based on instance state including the state of its incoming links. A target activity executes only if at least one of its incoming links evaluates to true and its join condition is true. A join condition failure, by default, generates a fault, and control is passed to the appropriate fault handler. This
fault, however, may be suppressed by setting the `suppressJoinFailure` attribute to true. In the latter case, the target activity is skipped, and all its outgoing links (if any) are set to false.

A generic use of the `flow` activity, including the use of a `link`, is shown in Figure 5.7. For brevity, not all messages are shown, and notably, transition and join conditions are omitted, and assumed to evaluate to true. The `flow` activity maps to a `flow` agent which waits for the preceding activity to finish (Sub1 in Figure 5.7), triggers the execution of each flow branch (Pub1 in Figure 5.7), and then waits for each branch to complete before triggering the subsequent activity.

Activities within a flow are first mapped to NIÑOS agents based on their associated transformation rules. For example, a `flow` activity agent will subscribe to and publish messages as outlined earlier. Then, each activity agent within a flow is augmented with the behavior described in the following paragraphs.

The first activity in each flow branch subscribes to the initiation of the flow (Sub2 in Figure 5.7), and publishes its completion as usual (Pub2 in Figure 5.7). Both activity2 and activity5 belong to this case in Figure 5.7.

Each link source activity publishes the transition condition of each outgoing link. In Figure 5.7, Pub3 indicates a true transition condition on activity2’s outgoing link. On the other hand, link targets subscribe to the status of their incoming links and the source activities associated with those links. For example, in Figure 5.7, activity6 subscribes to Sub4, and publishes Pub4 when it has completed successfully. A target activity that does not execute, due to a false join condition, publishes that it has skipped the execution of the activity. A successor activity to a link target must, therefore, subscribe to both the execution or suppression of its predecessor. In Figure 5.7, activity7, for example, would subscribe to Sub5 and publish Pub5 upon completion. Notice that the use of the `composite subscriptions` feature in Sub4 and Sub5 offloads the detection of event correlation patterns to the PADRES publish/subscribe layer, simplifying the work of the
activity agents.

All other activities publish and subscribe as usual, and do not change their behavior as a consequence of belonging within a flow.

Note that the cases above are not mutually exclusive, and an activity may be required to behave according to multiple descriptions. For example, an activity may be both the first activity in a flow branch and the target of a link, or may be both a source and target of (different) links.

Other activities

The mappings for the basic BPEL activities from Table 2.2 are relatively straightforward. For example, the reply activity subscribes to the successful completion of its predecessor activity, and publishes the reply message along with any variable updates. The fault activity, likewise, subscribes to the completion of its predecessor activity and publishes a fault message. The mapping of the sequence structured activity is also routine compared to the other activities described above. Each activity within a sequence simply subscribes to its predecessor’s completion, and publishes its own completion status.

BPEL variables

Activities within a BPEL process instance share data by means of variables, which are global within a process instance. NIÑOS supports two mechanisms to support BPEL variables.

The first mechanism maintains variables in a distributed manner. Every activity that modifies a variable publishes a VARIABLE_UPDATE message with the new value. Any activity that needs to read a variable issues a subscription for these update messages and caches this information locally. In this scheme, each activity agent independently maintains the variable value, and in the case of a sequential process, the value will be consistent across all activities.
A second mechanism addresses the issue of concurrent accesses to variables as is possible with activities executing in parallel flows in a process. In this case, a variable agent is used to maintain consistent variable values, and synchronize accesses to variables. Adopting standard distributed locking techniques, activities that read or write to variables must first acquire a read or write lock, respectively, from the variable agent and then retrieve the current variable value from the variable agent. The variable agent supports concurrent reads but exclusive writes. We plan in future work to explore the use of distributed locking algorithms that support greater concurrency and efficiency.

The variable agent mechanism can always be used, while distributed VARIABLE_UPDATEs are guaranteed to operate correctly only when variables are not accessed concurrently. Since it is straightforward to distinguish the potentially concurrent and sequential portions of a BPEL process, the process transformation is able to use the distributed VARIABLE_UPDATE mechanism in sequential parts of the process, but revert to variable agents in concurrent portions.

The visibility of variables by activities in different scopes is well-defined in the BPEL specification, and can be determined and resolved during process transformation. For example, activities would only issue subscriptions for updates to variables declared within their own or ancestor scopes. Other activities, for whom these variables are not supposed to be visible would not subscribe to and hence would not receive these variable updates.

### 5.1.4 Process deployment

The result of process transformation is a set of subscription, advertisement and activity information messages representing the BPEL activities in a business process. The goal of process deployment is to install an activity at a particular agent by sending the advertisements, subscriptions and the activity information generated from the transformation phase to available activity agents in the system.

Exploiting the publish/subscribe paradigm, the process manager wraps the above
messages inside envelopes compliant with the publish/subscribe language and sends them to activity agents. The envelopes are agent control publications with class `AGENT_CTL`, and contain the information that the manager wants to deliver to an agent, and the identifier of the particular agent in the `agentID` predicate. Activity agents receive the control messages by subscribing to `AGENT_CTL` messages addressed to themselves. Upon receiving an envelope, an agent unwraps the enclosed message and issues the messages as its own subscriptions or advertisements, as shown in Figure 5.8.

The process of installing an activity at an agent consists of five steps. First, a set of subscription, advertisement and activity information messages are generated from a business process definition file during the process transformation phase. Second, the messages are wrapped in an envelope as a field of an `AGENT_CTL` publication. Third, the publish/subscribe broker network delivers the `AGENT_CTL` publications to the addressed agents. Fourth, the agent extracts the subscription, advertisement, and activity information messages from the `AGENT_CTL` message. For instance, Pub1 in Figure 5.8 is an agent control publication wrapping a subscription for the `while` agent. Finally, the agent processes the messages based on the `command` field which has three possible values: `subscribe`, `advertise` or `activityinfo`. The `subscribe` command causes the agent to subscribe to the subscription specified in the `content` field, and not surprisingly, the `advertise` command causes the agent to advertise the advertisement contained in the `con-
tent field. Whereas subscriptions and advertisements describe the activity dependency of a process, the activityinfo control message contains information needed by an activity agent to execute the activity, such as the Boolean looping condition for a while activity. At this point, the business process is deployed and each agent is ready for execution.

We emphasize that after a BPEL process has been transformed into advertisements, subscriptions and activityinfo messages, there is much flexibility in the activity agents where these messages are installed. Furthermore, the provisioning of the quantity and types of activity agents can itself be arbitrary and accommodate to system requirements. For example, Figure 5.9 shows a scenario where organizations A and B decide to collaborate in hosting a business process. Each organization administers its own PADRES federation, and decides on which set of activity agents to provision. Notice that there may be multiple agents of the same type. Such replication of activity agents allows greater flexibility during process deployment, provides more resources with which to balance and support greater loads, and supports redundancy in case of failures. The BPEL process...
in Figure 5.9 may be deployed to the activity agents as annotated in the figure. Notice that regardless of the complexity of the network architecture, activity agents are simply identified by their location-independent address in the PADRES network, and the deployment of a BPEL process to activity agents proceeds exactly as above. As elaborated in Section 5.1.6, the ability of the PADRES content-based publish/subscribe layer to address components in the system by their network- and location-independent name is key to managing the complexity of arbitrarily elaborate deployments.

While organizations that wish to participate in the execution of a BPEL process must administer a PADRES/NIÑOS deployment, it remains possible to invoke processes hosted by other organizations (see Organization D in Figure 5.9) that expose their processes as Web services. Furthermore, BPEL processes executed by the distributed NIÑOS system can be invoked by outside clients (see Organization C in Figure 5.9). The scenario in Figure 5.9 illustrates the flexible deployment options available to organizations in terms of the distribution of the NIÑOS execution engine, and interactions with business partners and clients. The determination of an appropriate or optimal deployment is driven by business policies and goals and is the subject of ongoing work.

5.1.5 Process execution

The activity agents attached to the publish/subscribe system are responsible for executing the activities in the process. They are both subscribers and publishers, subscribing to activity completion events from predecessor activities and publishing events to notify their successor activities. The dual roles enable them to exchange messages within the publish/subscribe messaging system, allowing coordinated execution of the business process.

A particular instance of a process is started by a NIÑOS service request, such as Pub2 in Figure 5.8, and is driven by activity completion events. Execution continues until all the activities defined in the process are finished. The process flow, or dependen-
cies between activities, is encoded in the interplay between subscriptions and advertisements, which determine the order of activity execution. Dependency subscriptions may be composite subscriptions, in which case matching is performed in the broker network, and agents are notified only when their execution conditions are fully satisfied. Detecting the execution condition in the PADRES broker network makes the activity agent a light-weight component in NIÑOS without significant processing or storage requirements. During execution, all the message routing is automatic and transparent to the process management layer.

5.1.6 Process management

Enterprises demand powerful facilities to control and monitor their business processes. Convenient management features are even more important in distributed architectures. We highlight a few management scenarios below, and describe how they are supported in NIÑOS.

NIÑOS provides a graphical monitoring interface to visualize the network topology, message routing, and distributed process execution, as shown in Figure 5.10. The monitoring itself is entirely based on publish/subscribe messages, making it possible, for example, to observe what others are monitoring.

Both real-time and historic process monitoring are supported by NIÑOS. Real-time monitoring is simple to achieve in NIÑOS due to the use of a content-based publish/subscribe infrastructure. The monitor, shown in Figure 5.10, which is itself a publish/subscribe client, subscribes to the execution information of a particular activity, allowing the monitor to know the execution status of a process. The expressive content-based publish/subscribe semantics allow the monitor to observe the status of individual activities, trace the execution of a particular process instance, or perform countless other queries, all without requiring additional instrumentation logic at the components being monitored. For example, the first ACTIVITY_STATUS subscription in Figure 5.10 al-
Enterprise applications also require probing the execution of completed processes, perhaps for auditing or analysis purposes. The PADRES infrastructure supports historic data access using subscriptions that unify the query for past and future events [51]. Along with PADRES’s composite subscriptions feature [54], both executing and previously executed process instances can be correlated and queried. For example, it is possible to monitor the status of new process invocations by users who invoked the process at least ten times yesterday.

Another management scenario is to trace the execution of process instances that exhibit some behavior. For example, the second set of ACTIVITY_STATUS subscriptions\(^1\) trace the invocations of every activity for those process instances whose activity\(^2\) failed. Examining the execution of these instance can help diagnose the failure or understand its consequences.

Advanced process control functions include suspending, resuming or stopping running process instances. The target instances can be specified by instance id, process id, or any constraints expressible by the publish/subscribe semantics. For example, the

\(^1\)We use a composite subscription in this case.
Figure 5.11: Example loan application process

AGENT_CTL publication in Figure 5.10 instructs all agents executing the PAYROLL process to suspend the execution of instance p001.

These functions are useful especially when processes need to be updated on-line. For example, a manager may suspend running process instances, dynamically update certain activities in the process (by sending modified subscription, advertisement, and activity-info envelopes to activity agents), and resume the instances. The agent-based execution in NIÑOS simplifies this task since only the agents corresponding to the modified activities need to participate in the process redefinition. The other activities can continue executing the process.

As mentioned earlier, the provisioning of multiple instances of the same activity agent type provides more process deployment choices, greater scalability potential, and the ability to redeploy the activities assigned to a failed activity agent. For example, in
Figure 5.9, if the receive agent provisioned by Organization A fails, the Receive0 and Receive1 activities can be redeployed to the receive agent provisioned by Organization B using the management features described above. While the mechanisms required to respond to failures are supported by NIÑOS, the automatic detection and correction of failures is left for future work. Towards this end, we have investigated failure resilience in the PADRES network layer [41], and formalized well-defined semantics for the mobility of activity agents [36].

5.1.7 Example

Consider a loan approval BPEL process in Figure 5.11. The process is triggered when a loan application is received. In order to avoid approving risky loans, the process invokes two external Web services that independently generate a credit report for the loan applicant. Only if both credit rating services deem the applicant to be credit worthy does the process approve the loan application.

Each activity in the BPEL process in Figure 5.11 is mapped to a NIÑOS agent, and table 5.1 details the advertisements and subscriptions issued by each agent, as well the publications for a sample run of the process. Some of the agents in the parallel branches of the process, such as the invoke2 and receive3 activities, are omitted from Table 5.1. Their messages would correspond to the messages issued by the corresponding activities in the other branch.

Although there is a flow activity in the BPEL process in Figure 5.11, there is no corresponding agent. Instead, the first activity in each branch of the flow are triggered when the final activity before the flow activity completes. Similarly, it is possible to eliminate the switch activity by having the first activities in each branch of the switch subscribe to their respective conditions directly. The switch activity is assigned to an agent in Table 5.1 to illustrate what its message would look like.
<table>
<thead>
<tr>
<th>Activity</th>
<th>Subscription</th>
<th>Advertisement</th>
<th>Sample Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>(trigger)</td>
<td></td>
<td>[class,eq,TRIGGER], [process,eq,&quot;loan&quot;], [IID,isPresent]</td>
<td>[class,TRIGGER], [process,&quot;loan&quot;], [IID,&quot;g001&quot;], &lt;&lt;CLAIM&gt;&gt;</td>
</tr>
<tr>
<td>receive1</td>
<td></td>
<td>[class,eq,ACTIVITY_STATUS], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;receive1&quot;], [IID,isPresent], [status,isPresent]</td>
<td>[class,ACTIVITY_STATUS], [process,&quot;loan&quot;], [activityName,&quot;receive1&quot;], [IID,&quot;g001&quot;], [status,&quot;SUCCESS&quot;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[class,eq,VARIABLE_UPDATE], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;receive1&quot;], [IID,isPresent], [variableName,eq,&quot;claim&quot;]</td>
<td>[class,VARIABLE_UPDATE], [process,&quot;loan&quot;], [activityName,&quot;receive1&quot;], [IID,&quot;g001&quot;], [variableName,&quot;claim&quot;], &lt;&lt;CLAIM&gt;&gt;</td>
</tr>
<tr>
<td>assign1</td>
<td></td>
<td>[class,eq,ACTIVITY_STATUS], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;assign1&quot;], [IID,isPresent], [status,isPresent]</td>
<td>[class,ACTIVITY_STATUS], [process,&quot;loan&quot;], [activityName,&quot;assign1&quot;], [IID,&quot;g001&quot;], [status,&quot;SUCCESS&quot;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[class,eq,VARIABLE_UPDATE], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;assign1&quot;], [IID,isPresent], [variableName,eq,&quot;approval&quot;]</td>
<td>[class,VARIABLE_UPDATE], [process,&quot;loan&quot;], [activityName,&quot;assign1&quot;], [IID,&quot;g001&quot;], [variableName,&quot;approval&quot;], &lt;&lt;RESULT&gt;&gt;</td>
</tr>
<tr>
<td>invoke1</td>
<td></td>
<td>[class,eq,ACTIVITY_STATUS], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;invoke1&quot;], [IID,eq,$X], [status,eq,&quot;SUCCESS&quot;]</td>
<td>[class,ACTIVITY_STATUS], [process,&quot;loan&quot;], [activityName,&quot;invoke1&quot;], [IID,eq,$X], [status,&quot;SUCCESS&quot;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[class,eq,VARIABLE_UPDATE], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;invoke1&quot;], [IID,eq,$X], [variableName,eq,&quot;claim&quot;]</td>
<td>[class,VARIABLE_UPDATE], [process,&quot;loan&quot;], [activityName,&quot;invoke1&quot;], [IID,eq,$X], [variableName,&quot;claim&quot;], &lt;&lt;CLAIM&gt;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[class,eq,VARIABLE_UPDATE], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;wsA&quot;], [IID,eq,$X], [variableName,eq,&quot;claim&quot;]</td>
<td>[class,VARIABLE_UPDATE], [process,&quot;loan&quot;], [activityName,&quot;wsA&quot;], [IID,eq,&quot;g001&quot;], [variableName,&quot;claim&quot;], &lt;&lt;RESULT&gt;&gt;</td>
</tr>
<tr>
<td>receive2</td>
<td></td>
<td>[class,eq,ACTIVITY_STATUS], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;receive2&quot;], [IID,isPresent], [status,isPresent]</td>
<td>[class,ACTIVITY_STATUS], [process,&quot;loan&quot;], [activityName,&quot;receive2&quot;], [IID,&quot;g001&quot;], [status,&quot;SUCCESS&quot;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[class,eq,VARIABLE_UPDATE], [process,eq,&quot;loan&quot;], [activityName,eq,&quot;receive2&quot;], [IID,isPresent], [variableName,eq,&quot;reportA&quot;],</td>
<td>[class,VARIABLE_UPDATE], [process,&quot;loan&quot;], [activityName,&quot;receive2&quot;], [IID,&quot;g001&quot;], [variableName,&quot;reportA&quot;], &lt;&lt;RESULT&gt;&gt;</td>
</tr>
<tr>
<td>assign2</td>
<td>switch1</td>
<td>assign4</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td><img src="image1" alt="Assign2 Diagram" /></td>
<td><img src="image2" alt="Switch1 Diagram" /></td>
<td><img src="image3" alt="Assign4 Diagram" /></td>
<td></td>
</tr>
</tbody>
</table>

**Assign2**
- **Class**: ACTIVITY STATUS
- **Process**: loan
- **Activity Name**: receive2
- **IID**: $X$
- **Status**: SUCCESS

**Switch1**
- **Class**: SWITCH CASE TRIGGER
- **Process**: loan
- **Activity Name**: assign2
- **IID**: $X$
- **Triggered Activity**: assign4

**Assign4**
- **Class**: SWITCH CASE TRIGGER
- **Process**: loan
- **Activity Name**: switch1
- **IID**: $X$
- **Triggered Activity**: assign4
5.1.8 Architectural benefits

There are several capabilities offered by the distributed NIÑOS execution engine that are either not present in traditional centralized processing architectures or are more difficult to achieve. This section points out some of these qualitative benefits, deferring the quantitative performance benefits to Section 5.2.

One benefit is that fine-grained monitoring of a running process requires no additional effort, and little overhead. Since process activities are triggered by ordinary publish/subscribe messages, it is possible to non-intrusively subscribe to these messages and make detailed observations of running processes such as an activity’s invocation rate, a branch’s execution probabilities, or a process’s critical path. The monitoring does not require adding any additional instrumentation code to the process, and the multicast message propagation in PADRES ensures that only the minimal number of additional messages are sent over the network.
The NIÑOS architecture also naturally supports cross-organizational processes, where portions of a process span different administrative domains. For example, the portion of a loan application process that accesses confidential customer credit information may be required to execute within the accounting department. In NIÑOS, the relevant activities can easily be deployed on the resources administered by the appropriate department.

Related to the above point, NIÑOS also supports both orchestration and choreography styles of process execution. In fact, in NIÑOS a BPEL orchestration is mapped into a choreography involving a number of agents.

The NIÑOS architecture supports the capability to modify portions of a process while it is still executing. For example, the processing logic of a particular activity can be changed by deploying a replacement activity agent, having the new agent issue the necessary subscriptions and advertisements, and have the original agent unsubscribe. It is even possible to modify the control logic of a portion of a process using the same techniques to insert a new process fragment into an existing process. Since the process is distributed, this process modification technique allows the remainder of the process to continue executing while one portion of it is being altered.

The NIÑOS execution engine exploits the PADRES complex event processing capabilities to offload certain process execution tasks to the PADRES broker network. For example, activities that are triggered by multiple publications issue a composite subscription for these publications. The publications that contribute to matching the composite subscription are collected and correlated in the broker network itself. This benefits the agents who can avoid processing cost of the correlation, and reduces network traffic by letting the broker network decide the optimal point to collect and correlate the publications.

The decomposition of a process into fine-grained components affords more precise control over load balancing or replication needs. For example, a single activity in a process may be the bottleneck that limits the processing time of the process. Instead of
replicating the entire process, only the bottleneck activity needs to be duplicated. The details of realizing this are out of the scope of this article, but are made possible by the distributed NIÑOS execution architecture.

The distributed execution of activities in a process is also potentially more scalable by taking advantage of available distributed resources. Furthermore, due to the fine granularity of the individual execution agents, the system is able to utilize even relatively small resources. For example, certain activities in a process may be very lightweight and the associated agent could be deployed on a relatively underpowered machine; it is not necessary to find a machine that can execute the entire process.

One potential critique of the NIÑOS architecture is that it requires each organization to deploy a federation of PADRES brokers. However, this is conceptually no different from a process choreography where multiple organizations collaborate to execute a business process. In such choreography scenarios, the process spans administrative domains and there is no centralized coordinator, perhaps because the organizations cannot agree on one trusted central entity. Instead, each organization administers its own process execution engine, with standards such as BPMN [102] and the family of Web service specifications facilitating the interoperability among the participants. In a similar manner, the brokers in the NIÑOS architecture can use messaging and event processing standards such as the Java Messaging Service (JMS), Advanced Message Queuing Protocol (AMQP), or WS-Notification allowing each organization to deploy their choice of technology. It should also be reiterated that it is perfectly sensible to deploy the NIÑOS architecture on a single machine and only add additional resources as required.

Many of the benefits of the NIÑOS architecture stem from the distributed nature of the execution engine, where a large process is broken down into fine-grained activities which are each executed by an autonomous agent.
5.2 Evaluation

This section quantitatively evaluates the distributed NIÑOS process execution architecture. In particular, it is compared to centralized and clustered architectures. A variety of parameters are varied to attempt to understand the conditions for which each architecture is well suited.

5.2.1 Experimental setup

NIÑOS is implemented in Java over the PADRES distributed content-based publish/subscribe system. The evaluations are conducted in a dedicated network of 20 machines, each equipped with 4GB of memory and 1.86GHz CPUs. In all the tests, in addition to the deployed activity agents, there is a process management client, and a service request client that invokes process instances. Since there are no accepted benchmarks in this field, the delivery service business process described in Figure 5.2 is used.

The centralized, clustered and distributed execution deployments are compared. In the centralized scenario, activity agents reside on the same machine, connecting to a single PADRES broker. This deployment serves as a baseline and emulates a centralized execution engine. The clustered scenario attempts to increase the scalability of the system by deploying multiple centralized engines. In the evaluations, two sets of activity agents are deployed, with each set of agents connected to a different broker. A load balancer directs requests evenly across the two process engines, each of which independently execute the requests. For the distributed scenario, a 30 broker network is deployed with the agents connecting to the various brokers. The broker network has a maximum diameter of six hops, and there are 12 edge brokers and 18 inner brokers, the latter of which have a degree between two and four.

In all three deployments, two Web service agents act as proxies to the two external Web services invoked by the process. Since the availability of external Web services is
independent of the architecture of the execution, in all three deployments, the number of Web services is fixed at two. Notably, even though multiple copies of activity agents are deployed for the clustered scenario, these agents still share the two external Web services.

The metrics of interest are the average process execution time and the average system throughput while varying the request rate, the delay of the external Web services, and the size of the messages. The process execution time is defined as the duration from the issue of a request by a client to the receipt of the corresponding response by the client, and the throughput is the number of process instances completed per minute. The default values for request rate, Web service execution time, and message size are 500/min, 100 ms, and 512 bytes, respectively.

### 5.2.2 Request rate

This experiment varies the process invocation rate, where each invocation generates a process instance, and measure the average execution time and throughput. As shown in Figure 5.12, for lower request rates, the centralized approach offers the best execution time, with improvements of 9% and 20% over the clustered and the distributed deployments, respectively. This is because the overhead of the workload balancer in the clustered approach and the communication overhead of traversing the broker network in the distributed setup are not negligible. When the request rates are higher than 300/min,
however, the clustered and the distributed approaches are faster, with 34% and 67% better execution times, respectively, over the centralized scenario at the highest request rate of 6000/min.

The throughput results in Figure 5.12 show that the distributed and clustered approaches, whose maximum throughput are similar, outperform the centralized one, with a roughly 49% increase in maximum throughput. Notice that for low request rates of around 200/min, the throughput of all the three approaches are almost the same as the request rate because none of the approaches reach their maximum throughput. The virtually identical throughput at higher request rates for the clustered and distributed deployments is because the external Web services invoked by the process are the bottleneck. Since there are no replicas of these Web services, and they are shared among the process instances, the Web services limit the maximum throughput.

Note that a Web service behaving as a throughput bottleneck does not imply it is also a latency bottleneck. For example, even if the processing rate of a Web service is bound, the additional time it takes to execute the remainder of the activities in the process will differ in the distributed and clustered architectures. Therefore, the latency and throughput results in Figure 5.12 are not inconsistent.
5.2.3 Web service delay

To better understand the effects of the external Web services on both process execution time and throughput, this experiment varies the Web service delay from 20 ms to 2 sec with two different request rates. With a lower request rate of 50/min, the results in Figure 5.13 show that, as expected, a longer Web service delay increases the average execution time for all three deployment scenarios. When the delay is small, the centralized approach performs the best by avoiding the communication overhead present in the other two approaches. On the other hand, when the Web service delay increases, the distributed approach performs the best, with 49% and 70% improvements in execution time over the clustered and the centralized scenarios, respectively.

A large Web service delay requires the execution engine to handle many concurrent process instances, which increases the memory and processing requirements on the system. The increased number of process instances are balanced among two clusters in the clustered scenario, resulting in up to a 41% improvement in execution time compared to the centralized approach.

With a much higher request rate of 1000/min, the results in Figure 5.14 show that the centralized approach performs the worst and the distributed deployment the best regardless of the Web service delay. Recall from Figure 5.12 that all three deployment scenarios achieved their maximum throughput at a request rate of approximately 1000/min. At
this point process instances begin to queue up at activity agents and Web services, and the contribution of this queueing delay on the process execution time dominates the communication overhead present in the clustered and distributed deployments. As in the case with a lower request rate, the clustered and distributed deployments disperse the request load among the additional resources available to them and achieve faster execution times than the centralized scenario. As well, the throughput results in Figure 5.14 are consistent with the observations from Figure 5.12, with the throughput benefits of the distributed and clustered deployments disappearing when the Web service delays are increased to a point where the Web service becomes a bottleneck.
5.2.4 Message size

The amount of data transferred between activities in a BPEL process may vary. This experiment investigates the impact of the inter-activity message size for the three deployment scenarios. The results in Figures 5.15 and 5.16 indicate that varying the message size, even with different request rates, has little effect on either the process execution time or the throughput in any of the three deployment scenarios. However, the distributed case performs the worst with a request rate of 50/min because of the communication overhead and performs the best with a request rate of 500/min because the request queueing times dominate the communication overhead.

When the request rate is low (Figure 5.15), all three deployment scenarios maintain a throughput that roughly equals the request rate. However, the centralized deployment becomes overloaded with higher request rates (Figure 5.16), with the clustered and distributed approaches achieving about 48% better throughput figures. As with latency, the results show that the communication and processing overheads of traversing a larger broker network is not significant with message sizes up to 256 kbytes.

Larger messages influence performance in two key ways: an increase in the network overhead when transmitting messages, and an increase in the computation overhead when processing messages. Now, it is not clear to what extent the stable results in Figures 5.15 and 5.16 are generalizable to WAN deployments with slower network links, but the compute resources of the machines in the experiment are not unreasonable in commodity hardware, let alone enterprise servers. Therefore, the observation that the processing overhead is largely independent of the message sizes evaluated is likely a more universal phenomenon. This is understood by noticing that a significant portion of the processing overhead is attributable to the publish/subscribe matching of the messages, and the PADRES matching engine we use only performs matching on the attributes and predicates in the message header; the payload is not processed. To exploit this, the BPEL process transformation in NIÑOS only encodes the process control flow details as publish/
subscribe attributes and predicates, and leaves the process data in the payload. In particular, VARIABLE_UPDATE publications include the variable’s name as a publish/subscribe attribute, but store the variable’s value in the message payload. In this way, variations in message size caused by variable data values do not significantly influence the publish/subscribe matching time.

5.2.5 Parallelism

As processes may exhibit different degrees of parallelism, this experiment compares two processes: one containing many activities with ten parallel branches, as shown in Figure 5.17, and another with the same number activities but with only two parallel branches. To isolate the effects of process parallelism, no external Web services are invoked by either process.

With the highly parallel process, the distributed deployment offers significantly better execution time performance as shown in Figure 5.18(a). When the request rate is less than 100/min, we observe a trend similar to Figure 5.12, where the distributed approach performs worse because of the additional network overhead. This is understandable since
higher request rates result in more activities executing in parallel, and more opportunities for the distributed deployment to take advantage of the additional resources available to it. At the highest request rates evaluated, the distributed scenario executed the parallel process 79% faster than the centralized approach, and the clustered deployment improved over the centralized one by about 71%.

With the more sequential process, on the other hand, the benefits of additional resources diminishes. It turns out that there is not enough parallelism in this particular process for the distributed deployment to benefit from, and the clustered approach actually achieves the best execution time. The results in Figure 5.18(b) indicate that at the request rate of 2000/min, the clustered approach is 64% faster than the centralized one, whereas the distributed deployment is only 42% better.

Since there are no Web service requests in the processes, which might limit the maximum throughput, the clustered approach has the best throughput with 25% and 41% improvement for the highly parallel and the less parallel processes, compared to the distributed approach.

According to the results, the distributed approach is better able to achieve low process execution times with processes characterized by many parallel flows and in situations
where the request rates are high. Otherwise, there is not enough parallelism to exploit the distributed resources available and the distribution overhead may actually impair the performance. In such situations the centralized or clustered architectures may perform better.
Chapter 6

Distributed activity placement

This chapter develops a distributed algorithm to make runtime decisions on where to deploy activities in a process. Section 6.1 describes a cost model and application framework for the activity placement algorithm, followed by extensions to these algorithms in Section 6.2. Section 6.3 then illustrates how the monitoring of an SLA can be modelled as a process and optimized by the same algorithms developed above. Finally, Section 6.4 presents an evaluation of these algorithms.

6.1 System architecture

A distributed execution engine provides a large number of possible deployment options. However, determining the strategic deployment of activities that minimizes some cost can be a labour intensive procedure that requires knowledge of the system resources and process characteristics. It may even be a futile exercise if either of these variables changes frequently. It is desirable for the system itself to determine an optimal placement of activities. To achieve this, we develop a cost model below to model the cost of a particular placement of activities. This model is used to compare different placement possibilities.

The cost model is a framework that consists of various factors that can influence the
activity placement decisions. Some cost factors are shown in Table 6.1 grouped into \textit{cost components}.

The first component is the \textit{distribution cost} which represents the overhead of distributing a process into small, fine-grained activities. This overhead can be expressed in terms of the bandwidth or latency of the inter-activity communication depending on the desired goal.

Another important cost component captures the resource usage of an activity at the engine it is executing. Factors here include the number of concurrent instances an activity is executing, the resource utilization (in terms of processor or memory) of an activity, and the complexity of the task the activity is executing.

The third cost component in Table 6.1 is the \textit{service cost} which represents the cost of calling external services. This includes the time to call the service (which is a function of the network conditions between the activity and service), and the execution time of the service (which depends on the particular service provider used to execute the desired service).

A \textit{cost function} based on various cost components can flexibly express different goals. The cost function specifies that an arbitrary weighting of the various cost components either meet a \textit{threshold} or should be \textit{minimized}. In the former case, the process is adapted only when the threshold is violated, while in the latter, process adaptation occurs whenever a more optimal placement is found. For example, Table 6.2 shows cost

<table>
<thead>
<tr>
<th>Component</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution cost</td>
<td>$C_{\text{dist}}$ (distribution overhead)</td>
</tr>
<tr>
<td>Message rate</td>
<td>$C_{d1}$</td>
</tr>
<tr>
<td>Message size</td>
<td>$C_{d2}$</td>
</tr>
<tr>
<td>Message latency</td>
<td>$C_{d3}$</td>
</tr>
<tr>
<td>Engine cost</td>
<td>$C_{\text{eng}}$ (execution overhead)</td>
</tr>
<tr>
<td>Load</td>
<td>$C_{e1}$</td>
</tr>
<tr>
<td>Resources</td>
<td>$C_{e2}$</td>
</tr>
<tr>
<td>Task complexity</td>
<td>$C_{e3}$</td>
</tr>
<tr>
<td>Service cost</td>
<td>$C_{\text{serv}}$ (service overhead)</td>
</tr>
<tr>
<td>Service latency</td>
<td>$C_{s1}$</td>
</tr>
<tr>
<td>Service execution</td>
<td>$C_{s2}$</td>
</tr>
<tr>
<td>Marshalling</td>
<td>$C_{s3}$</td>
</tr>
</tbody>
</table>

Table 6.1: Cost model components
### Criteria

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Cost function mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>3s response time</td>
<td>$C_{d1} + C_{d3} + C_{e3} + C_{serv} &lt; 3$</td>
</tr>
<tr>
<td>Optimize bandwidth</td>
<td>$\min(C_{d2})$</td>
</tr>
<tr>
<td>Minimize server energy use</td>
<td>$\min(C_{e4})$</td>
</tr>
<tr>
<td>Balance energy, bandwidth costs</td>
<td>$\min(($3 \text{ per energy unit})C_{e4} + ($4 \text{ per bandwidth unit})C_{d2})$</td>
</tr>
</tbody>
</table>

Table 6.2: Examples of cost functions

functions that constrain process response times to three seconds, and that minimize the network overhead of a process.

#### 6.1.1 Distributed execution

As discussed earlier, we take a distributed approach to the execution of business processes, whereby individual tasks in a process are executed by autonomous engines which collaborate to execute the original process. The activities execute within a distributed execution engine whose architecture is shown in Figure 6.1.

In the spirit of the distributed nature of the system, each engine is autonomous in deciding whether to host an activity or move it to another engine. These decisions are based on the cost function associated with the process. Notably, there is no centralized component that is used to gather statistics, or to make activity placement decisions.

The distributed execution engine shown in Figure 6.1 consists of a core Activity Manager that provides support services for activities to collaborate among one another to execute a particular business process. A Candidate Discovery component is used to find other execution engines in the system. In the current implementation, the discovery component returns all neighbours of an engine within a given radius. The discovered candidates are periodically probed by the Engine Profiler components to gather various statistics. The Redeployment Manager computes the cost function for each activity executing in the engine, and determines if a more optimal placement of the activity is available among the known candidate engines. Finally, activities that are to be moved are redeployed using the Atomic Redeployer component which is responsible for ensuring that the movement of the activity does not affect the execution of the process. Briefly, the
redeployer pauses the triggering of new instances of the activity, transfers the activity state to the new engine, rebinds the activity to its successors and predecessors in the process, and resumes the execution of the activity.

### 6.1.2 Profilers and estimators

For every metric supported by the system, the execution engines contain an associated activity profiler, engine profiler, and estimator.

In the following discussion, we suppose that a set of activities $A$ are deployed across a set of execution engines $E$, with each activity $a_i \in A$ deployed on exactly one engine $e_j \in E$. The set of activities deployed at a given engine $e_j$ is $A(e_j)$, and $E_c(e_j)$ refers to the neighbourhood of candidate engines that engine $e_j$ is aware of.

Let $M$ denote the set of all supported metrics. Then, for each metric type $M_k \in M$, there is an activity profiler $AP_k$, an engine profiler $EP_k$, and an estimator. Furthermore, $s_k(a_i)$ and $p_k(e_j)$ denote the information about an activity $a_i$ and execution engine $e_j$, respectively, relevant to metric $M_k$. 
Activity profiler

The activity profiler records statistics $s_k(a_i)$ on a given activity $a_i$ that are relevant for a given metric type $M_k$. These statistics may consist of an arbitrary set of values, including histograms on these values. In the schematic of the activity profiler component depicted in Figure 6.2, the synchronous input to and output from the component are drawn on the left and right, respectively, and the asynchronous inputs to the component are marked above.

For example, an activity profiler for a message bandwidth metric type would may record the average message rate between an activity $a_i$ and its predecessor and successor activities. This information is necessary to compute the bandwidth cost of hosting activity $a_i$ at a given engine. Likewise, an activity profiler for a memory consumption metric type would maintain the memory consumption patterns of the activity. Depending on the capabilities of the platform, an activity profiler for an energy use metric may record the rate and distribution of an activity (to determine how frequently the system can take advantage of hardware sleep states), or how CPU or I/O bound an activity is (to determine the impact of techniques such as dynamic CPU frequency scaling, or disk rotation speeds).

The profiler needs to instrument the execution of activities at an engine. This may entail registering callbacks so that the profiler is aware of various lifecycle stages of the activity, or require probing the activity behaviour. The details of how the profiling is
carried out is specific to the profiler requirements and execution engine capabilities.

For an engine \( e_j \), the activity profiler must maintain a constant amount of statistics for the set of local activities \( A(e_j) \) hosted by the engine, resulting in a memory complexity of \( O(|A(e_j)|) \). Retrieving this information is simply a table lookup and requires constant time.

**Engine profiler**

![Diagram of engine profiler](image)

Figure 6.3: Engine profiler

The engine profiler maintains information \( p_k(e_j) \) of a remote engine \( e_j \) necessary to estimate the *metric* cost of hosting a given activity at engine \( e_j \). The profiler is responsible for periodically retrieving this information from remote engines.

For example, an engine profiler for a latency metric type would periodically compute and store the network latency between the local engine and remote engine \( e_j \). A profiler for an energy use metric would need information about the dynamic energy scaling characteristics of the hardware. In a cloud environment, for example, a server may enter a sleep state when there are no activities deployed on it, and may be able to enable CPU cores as it handles more activities. In this case, the energy profile of the engine may be a staircase function of the number of activities deployed on it.

At a given engine \( e_j \), the engine profiler must maintain a constant amount of information for each candidate remote engine known to \( e_j \), denoted \( E_c(e_j) \). This results in a memory complexity of \( O(|E_c(e_j)|) \). This information can be looked up in constant time lookup.
It should be noted, however, that the complexity of computing the remote engine profile varies with each engine profiler. For example, the hop count metric requires a message complexity proportional to the path length between $e_j$ and the remote candidate engine.

**Estimator**

The metric estimator is used to compute an estimate of the metric cost $c_k(a_i, e_j)$ of hosting an activity $a_i$ on engine $e_j$. Internally it makes use of $AP_k$ and $EP_k$, the associated activity and engine profilers, respectively.

While the estimator itself is stateless, it makes use of the state maintained by the activity and engine profilers, so it requires $O(|A(e_j)| + |E_c(e_j)|)$ memory. The computational complexity of the estimator depends on the particular metric type. For example, a latency estimator must compute the network latencies between the current engine and those engines where an activity’s predecessor and successor activities are deployed.

To avoid unnecessary work, a profiler is enabled only if there is a locally hosted activity whose cost function depends on the metric computed by the estimator. For example, if there is an activity whose cost function is to minimize message latency, the latency estimator would be enabled but not the bandwidth estimator. Each estimator is also provided with additional information necessary to perform the estimations. For example, the latency estimator is given a set of the predecessor and successor engines $e_n$ associated with relevant activities.
Example: hop-count engine profiler

This section describes an example of an engine profiler, namely one that efficiently computes the distance between engines.

As an activity communicates with its predecessor and successor activities, the engine discovers the paths among these activities. This is done by having intermediate brokers in the overlay network append their identifiers to messages that traverse through them. This is a relatively light-weight operation and incurs virtually no processing or network overhead. Therefore, for an activity $a_i$ executing on engine $e_i$ communicating with a predecessor activity $a_p$ executing on engine $e_p$, engine $e_i$ knows the brokers in the overlay path from engine $e_p$ to $e_i$. This path is denoted as $e_p \rightsquigarrow e_i$, and a sub-path between two engines $e_x$ and $e_y$ that exist on this path is denoted as $e_p \overset{x}{\rightsquigarrow} e_i$.

Note that an activity $a_p$ is considered a predecessor of an activity $a_i$ if there is a transition in the control flow of the process from $a_p$ to $a_i$. Similarly, activity $a_s$ is a successor of $a_i$ if there is a transition from $a_i$ to $a_s$.

Given the paths between an activity $a_i$ and its neighbors (predecessors and successors), the hop-count profiler can compute the paths between a candidate engine $e_c$ and the neighbors of $a_i$. This is done without having to probe the paths between these engines and so requires no additional overhead. The following mutually exclusive and exhaustive cases are considered.

1. $e_c \in e_p \rightsquigarrow e_i$. In this case, the candidate engine is within the path between the predecessor and given engines. Therefore, $e_p \rightsquigarrow e_c = e_p \overset{p}{\rightsquigarrow} e_i$.

2. $e_p \in e_c \rightsquigarrow e_i$. Here, the predecessor engine is within the path between the candidate and given engines. Therefore, $e_p \rightsquigarrow e_c = e_i \overset{p}{\rightsquigarrow} e_c$.

3. If neither case above applies, then $e_p \rightsquigarrow e_c = e_p \overset{p}{\rightsquigarrow} e_i \overset{y}{\rightsquigarrow} e_c$, where $e_y$ is an engine that (1) exists in the path between the predecessor and given engines as well as the path between the given and candidate engines, and (2) minimizes the path...
between the predecessor and candidate engines. More formally, \( e_y \in e_p \leadsto e_i \) and \( e_y \in e_i \leadsto e_c \) such that \( \not\exists e_z \) where \( e_z \in e_p \leadsto e_i \) and \( e_z \in e_i \leadsto e_c \) and \( e_y \leadsto e_i \) is a subsequence of \( e_z \leadsto e_i \).

Corresponding cases can be used to compute the paths between candidate and successor engines. Note that the above computations assume an acyclic overlay topology. In a cyclic topology, these algorithm would only compute a possible distance between engines, not necessarily the shortest path.

It may not be obvious, but the last case above is actually general enough to subsume the other two. However, the other cases are still useful in order to simplify the path computations.

### 6.1.3 Redeployment manager

![Redeployment manager diagram](image)

The Redeployment Manager maintains the following information for each activity \( a_i \) the engine is currently hosting: the cost function \( f(a_i) \) associated with the activity, a running average of the cost \( c(a_i, e_j) \) imposed by the activity if it were hosted at engine \( e_j \), and the engines where activity \( a_i \)'s predecessors and successors are hosted. For convenience, the cost of deploying \( a_i \) at the current engine is denoted as \( c(a_i) \). The cost \( c(a_i) \) has two different interpretations depending on the type of cost function. For threshold functions, \( c(a_i) \) is the accumulated cost by all activities from the beginning of the process to the current activity, whereas for minimum or maximum cost functions, \( c(a_i) \) is the local cost of the activity. An example should make the reason for the difference clear. Consider a cost function to minimize the message latency of a process: \( \min(c_{d3}) \). In this
case, the local cost of an activity is the latency of communicating with its predecessors and successors. To minimize the overall latency, each activity should attempt to minimize its local latency cost. On the other hand, for a cost function that requires the message latency to stay below a threshold, such as \( c_{d3} < 10 \), it is necessary to keep track of how much each activity contributes to the overall latency of the process. The local latency cost of each activity must be accumulated as the process flow executes. To achieve this, for activities associated with threshold cost functions, messages between activities are annotated with the accumulated cost.

The running average of the cost \( c(a_i, e_j) \) of an activity is computed and maintained based on information from various profilers and estimators. For example, consider an activity \( a_i \) hosted on engine \( e_i \), with predecessor activity \( a_p \) and successor activity \( a_s \), hosted on engines \( e_p \) and \( e_s \), respectively. The local message latency cost for an activity \( a_i \) is \( \text{Latency}(e_p, e_i) + \text{Latency}(e_i, e_s) \), where \( \text{Latency}(e_p, e_q) \) is the latency between engines \( e_p \) and \( e_q \) as determined by the latency estimator.

The redeployment manager recomputes the costs \( c(a_i, e_j) \) when one of two conditions occurs. When the activity \( a_i \) executes (including when it sends and receives messages with its successors or predecessors), the cost for the current engine \( c(a_i, e_{\text{current}}) \) is updated. Likewise, when an estimator updates a metric that is included in the cost function associated with \( a_i \), the cost of hosting the activity at the candidate engine whose metric was just updated is recomputed. To facilitate the latter case, each estimator is given a list of activities which are relevant to the metric the estimator is computing. Therefore, the estimator will only notify the redeployment manager when necessary.

An update of the cost \( c(a_i, e_j) \) may reveal a better placement for activity \( a_i \). Every update to the cost of an activity initiates a call to the \( \text{CheckDeployment}(a_i) \) function to find a more optimal deployment. The \( \text{CheckDeployment}(a_i) \) algorithm differs based on the cost function associated with activity \( a_i \), and is outlined in Algorithm 3.

If the cost function is to be minimized, the algorithm finds the engine \( e_{\text{min}} \in E \) such
Algorithm 3: Determine deployment of activity \( a_i \) currently hosted at engine \( e_c \).

\[\textbf{Input:} \ a_i \leftarrow \text{the activity to consider}\]

1. \textbf{if} \( f(a_i) \) is minimize type \textbf{then}
   
   2. \( e \leftarrow \text{findBestLocation}(a_i); \)
   
   3. \( \text{benefit} \leftarrow c(a_i) - c(a_i, e); \)
   
   4. \( \text{resident} \leftarrow \text{NOW} - \text{arrival}(a_i); \)
   
   5. \textbf{if} \( \text{benefit} > T_{\text{benefit}} \land \text{resident} > T_{\text{resident}} \) \textbf{then}
      
      6. \( \text{redeploy}(a_i, e); \)
   
   7. \textbf{end}

8. \textbf{else if} \( f(a_i) \) is threshold type \textbf{then}
   
   9. \textbf{if} \( f(a_i) \) is false \textbf{then}
      
      10. \( e \leftarrow \text{findBestLocation}(a_i); \)
      
      11. \( \text{benefit} \leftarrow c(a_i) - c(a_i, e); \)
      
      12. \( \text{resident} \leftarrow \text{NOW} - \text{arrival}(a_i); \)
      
      13. \textbf{if} \( \text{benefit} > T_{\text{benefit}} \land \text{resident} > T_{\text{resident}} \) \textbf{then}
          
          14. \( \text{redeploy}(a_i, e); \)
      
      15. \textbf{end}

16. \textbf{if} \( \text{resident} > T_{\text{resident}} \land e = e_c \) \textbf{then}
      
      17. \( \text{applyBackPressure}(a_i); \)
   
   18. \textbf{end}

19. \textbf{end}

20. \textbf{end}

that \( c(a_i, e_i) \) is minimized across all \( e_i \in E \) where \( E \) is the set of known candidate engines. Activity \( a_i \) is then moved to engine \( e_{\text{min}} \). To avoid frequent redeployment, an activity is redeployed only if the improvement in the cost exceeds a given threshold \( T_{\text{benefit}} \) and if the activity has not been redeployed for some time duration \( T_{\text{duration}} \). The values \( T_{\text{benefit}} \) and \( T_{\text{duration}} \) have system wide defaults that may be overridden by each cost function.

If the cost function associated with activity \( a_i \) is a threshold function, a check is made to see if the accumulated cost \( c(a_i) \) exceeds the threshold. If the cost is still within the threshold, nothing further is done. Otherwise, the \( \text{CheckDeployment}() \) function finds the engine \( e_{\text{min}} \) that results in \( \min_{e_i \in E} c(a_i, e_i) \), and redeploy activity \( a_i \) to engine \( e_{\text{min}} \). Now it may be that \( c(a_i, e_{\text{min}}) \) still exceeds the cost function threshold, in which case a message is sent to the predecessor engines of activity \( a_i \) to force them to redeploy. Notice that these predecessors would not have normally chosen to redeploy because their accumulated cost
is still within the threshold. This “back pressure” by activities to force a redeployment of their predecessors will occur repeatedly as long as the optimal placement of the activity is not sufficient to satisfy the cost function threshold.

Cost model

![Cost model diagram](image)

The cost model computes the cost \( c(a_i, e_j) \) of hosting activity \( a_i \) on engine \( e_j \). Internally it uses the estimators for each metric type. The total memory consumption of these estimators at engine \( e_c \) is \( O(|M(e_c)|(|A(e_c)| + |E_c(e_c)|)) \), where \( M(e_c) \) is the set of metric types understood by engine \( e_c \). Computing the cost \( c(a_i, e_j) \) requires looking up values in the estimators and evaluating the cost function, which requires \( O(|M(e_c)|) \) time.

6.1.4 Atomic redeployer

The actual movement of an activity \( a_i \) from engine \( e_1 \) to engine \( e_2 \) as determined by the Redeployment Manager is carried out by the Atomic Redeployer. The challenge is to move an activity without disrupting the process and to ensure that failures during movement do not leave the system in an inconsistent state. The movement is modelled as a transaction consisting of a single \( \text{move}(a_i, e_1, e_2) \) operation. If the transaction aborts—perhaps because \( e_2 \) refused to accept activity \( a_i \)—the activity must remain at engine \( s_1 \), otherwise if the transaction commits, the activity must be instantiated at engine \( e_2 \) and deallocated from engine \( e_1 \). In either case, the predecessors and successors of activity \( a_i \) should be unaware of the movement, no messages must be lost, and each message
should be delivered to either the activity instance at $e_1$ or at $e_2$ but not both. These requirements and others have been formalized in detail, and the algorithms to achieve atomic movement satisfying these requirements have been developed, their correctness proven, and their performance quantified [36].

6.2 Extended techniques

The basic redeployment algorithm described above may not perform well under certain conditions, primarily due to the distributed nature of the planning decisions. This section outlines a number of techniques that overcome these limitations.

6.2.1 Fluctuations

Since each engine independently carries out redeployment decisions concurrently based largely on local information, it is possible that two or more engines repeatedly alternate among a set deployment decisions.

A number of relatively simple constraints are applied during the redeployment algorithm to reduce the likelihood of such alternating decisions.

Two thresholds are used to avoid those redeployments that are likely to result in unstable behaviour. A benefit threshold $T_{benefit}$ is used to only move activities that will provide a non-trivial benefit, and a duration threshold $T_{duration}$ ensures that an activity is hosted at an engine for a sufficiently long time before it can be redeployed. In addition, a random jitter is applied to $T_{duration}$ so that engines do not redeploy activities in lock-step. This increases the chance of an engine observing the effect of the redeployments made by another engine before it makes its own decision.

In addition, during each planning phase, only the top $k$ most beneficial redeployment opportunities are carried out. Again, this allows the effect of a few redeployments to be observed before more decisions are made.
6.2.2 Activity set redeployment

There are cases where the distributed redeployment algorithm may get stuck in a locally optimal configuration. One reason for this is that the algorithm considers the deployment of each activity individually. By considering the possibility of moving sets of activities, it may be possible to escape some of these local minima cases. The experiment in Section 6.4.4 illustrates this point.

Algorithm 3 can be extended so that it estimates the SLA cost function when sets of activities are redeployed. However, given \( n \) activities deployed at an engine, there are \( 2^n - 1 \) subsets of these activities, and it is not feasible to estimate costs for each set. Instead, the algorithm restricts itself to activities that are contiguous in the workflow, and only to sets with no more than \( T_{\text{maxset}} \) activities. These heuristics still leave the algorithm vulnerable to locally optimal configurations, but substantially reduce the search space of the algorithm and make the problem tractable.

6.2.3 Configuration sensitivity

The redeployment manager makes activity placement decisions using activity and engine profiles gathered over a window of samples. For example, the activity profiler may compute the average number of invocations of an activity over a time window of duration \( \theta \). The choice of \( \theta \) represents a tradeoff between the responsiveness and accuracy of the profiler and ultimately of the redeployment decisions as well. Typically, a larger \( \theta \) will, by gathering more samples, result in a more accurate reading of the profiled values. The larger sample window, however, also means that it will take longer for changes in the observed metrics to be reflected in the averaged results, and hence activity redeployment decisions may lag the optimal movement time. On the other hand, with a small \( \theta \), the profilers will be more responsive to changes in the sampled values, but are susceptible to short term perturbations in the samples. This can be a problem as it may result in
frequent activity movement decisions as the system reacts to transient disturbances in
the environment or process flow. An example of this phenomenon is demonstrated in the
evaluations in Section 6.4.6.

To increase the robustness of the deployment decisions, and tolerate short-term per-
turbations in the system, the redeployment algorithm is extended to record the dis-
tributions of observed values, and to only act on those values that occur with a high
probability.

In particular, the activity profiler maintains a histogram of statistics \( H_{a_k}(a_i) \) on ac-
tivities instead of a running average of these statistics over some window. The domain
of these histograms is a predefined set of ranges over the value being measured. For
example, for a metric that measures the message latencies, it would store a histogram of
the number of messages whose latencies fall into certain latency ranges. Similarly, the
engine profiler records a histogram of information \( H_{p_k}(e_j) \) about known engines in the
system. The metric estimator then computes a joint histogram \( H_{c_k}(a_i,e_j) \) on the two
histograms accumulated by the activity and engine profilers.

Finally, the cost model applies the cost function to this distribution of metric costs
to compute a distribution of total cost \( H_c(a_i,e_j) \) of deploying an activity \( a_i \) at engine
\( e_j \). Now, this histogram can be used to determine a more robust estimate of a particular
deployment alternative. For example, by looking up the 95\% value in a cumulative
distribution of \( H_c(a_i,e_j) \), the algorithm considers the cost of the deployment decision
under all but the most unlikely set of conditions.

6.3 Process monitoring case study

This section illustrates an end-to-end example that demonstrates the methodology de-
veloped in this thesis. In this case study we express system monitoring as a process.

Monitoring a business process to observe SLA violations is typically carried out by
collecting traces of the execution of a process and processing these traces in a centralized location. We observe, however, that the processing of these traces itself resembles a workflow.

We have devised a procedure to systematically map an SLA into a workflow of monitoring activities that collaborate to observe violations of the SLAs. The generic structure of this workflow is shown in Fig. 6.7. In this case, each activity maps to an artifact in the SLA, such as the individual XML elements in the WSLA language specification [38]. The Metric activities observe raw system and application trace events or aggregate the output of other Metric activities. An SLO activity computes a service level objective expression specified in the SLA, such as constraints on the process throughput or energy budget consumed. Finally, the Action activity reports any SLO violation. Additionally, Scope activities can constrain the SLA computations to a subset of process instances, such as those associated with users in a particular geographic location, and Exclusion activities can selectively disable SLA computations, such as during periods of planned system maintenance. Notice that as with the distributed process execution, the SLA
workflow activities coordinate themselves by emitting and consuming events.

The consequence of this design is that the SLA monitoring workflow itself can have an SLA applied to it. For example, we can apply a cost function to minimize the monitoring message overhead which may result in certain activities automatically relocating close to the busy portions of the process being monitored. Likewise, a cost function to instead minimize latency would deploy the SLA activities to report violations quickly.

Notice that we have used the distributed execution engine to carry out a monitoring “process”. The same principles used to optimize the deployment of a business process according to some SLA goals can now be applied to the monitoring as well. Evaluations in Sec. 6.4.5 verify the effectiveness of this approach.

6.4 Evaluation

This section evaluates the algorithms described in this thesis as implemented in the open-source PADRES messaging platform. The experimental testbed consists of a cluster of 1.86 GHz Xeon machines with 4 GB of RAM. The default topology of PADRES messaging brokers and execution engines is shown in Figure 6.8. In the deployment, each pair of broker and execution engine is assigned a dedicated machine in the cluster.

Each experiment that follows is designed to demonstrate one or more aspects of the algorithm, or illustrate a weakness of the algorithm that needs to be addressed.
In the experiments, the SLA is to minimize the message traffic in the network. Therefore, the metric measured is the number of message hops, where each overlap link contributes one hop.

### 6.4.1 Process hotspots

The first set of experiments use the workflow shown in Figure 6.9, whose control flow includes parallel execution and looping constructs. For any given execution of this workflow that arrives at activity $E$, the next transition in the workflow goes to activity $D$ with probability $p$, and activity $F$ with probability $1 - p$. Similarly, the transition from activity $F$ to activity $E$ occurs with probability $q$, and from $F$ to $G$ and $H$ with probability $1 - q$. The default values of $p$ and $q$, are 0.9 and 0.1, respectively.

The activities are initially deployed to the engines as shown in Figure 6.8. Furthermore, activities $D$ and $F$ are pinned to their respective engines and cannot be redeployed to other engines. In practical scenarios, such constraints may arise from security or administrative concerns over where activities may be deployed.

The activities are initially deployed as shown in Fig. 6.8, with activities $D$ and $F$ pinned to their respective engines. Such constraints may arise from security or administrative concerns over where activities may be deployed. Figure 6.10 shows a scatter plot of the message hops for instances of the workflow. An instance of the workflow is triggered every second, and each point in the plot represents the number of message hops traversed during the execution of a particular instance. Due to the looping probabilities of the process, the message hops vary by instance. To better observe trends, a moving average of the message hops is also shown in the plot.
In Fig. 6.10(a), the activities remain at their initial deployments and the cost of each instance hovers around twenty to thirty message hops. With the basic redeployment algorithm, however, the activities are redeployed and a lower cost is achieved as shown in Fig. 6.10(b). This figure also indicates when the redeployment occurs with a vertical line; the height of this line has no meaning. The algorithm takes about thirty seconds to profile the activities and engines, plan a more optimal deployment, and move the activities. Referring back to Fig. 6.8, what occurs is that activities A, B, and C are moved to engine 1, and activities G, H, and I to engine 7. In addition, activity E moves to engine 1 to be closer to activity D; activities D and E have a tight loop in the workflow and co-locating them brings about significant savings. The resulting deployment in steady state requires only about 10% of the message hops as the static deployment.

This experiment demonstrates that it is not necessary to manually deploy activities in a strategic manner — a possibly complex task — but the system can configure itself.

### 6.4.2 Varying process hotspots

In a variation of the above experiment, we now vary the branch probabilities during the experiment. The initial values of probabilities \( p \) and \( q \) in the process in Figure 6.9 are initially 0.9 and 0.1, respectively, but halfway through the experiment, the values are
reversed to \( p = 0.1 \), and \( q = 0.9 \). In real applications, the control flow patterns may vary due to factors such as the time of day or change in the inputs to the workflow.

Figure 6.11 compares a static deployment with the basic redeployment algorithm. We observe in Figure 6.11(b) how the change in the workflow branch characteristics at around 100 seconds causes a sharp increase in the workflow cost, but within about thirty seconds the profilers detect this change, and the redeployment algorithms move the activities to restore a low cost deployment. Specifically, activity \( E \) is moved to engine 7 so as to minimize the cost of executing the new hotspot between activities \( E \) and \( F \).

It is interesting to note that immediately after the branch probabilities are changed, the dynamic algorithm in Figure 6.11(b) incurs a higher message hop cost than the static case in Figure 6.11(a) for the short period before another set of redeployments takes place. This is because the dynamic algorithm had already optimized deployments for the initial control flow patterns which, in this case, turns out to be a poor choice for the modified control flow patterns.

A notable point about this experiment is that because the conditions of the system change over time, there is no optimal static deployment. Dynamic reconfiguration is required to achieve the best results.
6.4.3 Convergence

This experiment shows the effects of a large candidate engine discovery radius. In the previous experiments, each execution engine is only aware of those engines one hop away in the overlay. As before, the process in Figure 6.9 is deployed to the engines in Figure 6.8, but activity $F$ is initially deployed to engine 8 and the remaining activities are assigned to engine 2. The purpose of this deployment is to observe how quickly activity $F$ moves to its optimal location at engine 2.

Figure 6.12(a) shows that it takes four movements and about 140 seconds for the system to stabilize at the optimal deployment. Observe that each redeployment decision moves activity $F$ by one hop, and results in a lower message hop cost. When the candidate radius is increased to two hops, we see in Fig. 6.12(b) that the system stabilizes after two movements in about half the time.

6.4.4 Local optima

The basic redeployment algorithm estimates the benefit of moving individual activities. We now compare this algorithm to one that considers moving sets of activities.

Consider the initial deployment of the process as shown in Figure 6.13. The basic redeployment algorithm can determine that activities $D$ and $E$ should be co-located. At
this point, moving either activity $D$ or $E$ individually anywhere else will result in an increase in the message hop cost, and so the system will make no further movements. In particular, it will not converge to a deployment where these activities also move to engine 1 to be closer to activity $F$.

Figures 6.14(a) and 6.14(b) show two runs of the system in which either engine 8 acts first to move activity $D$ to engine 7, or engine 7 first moves activity $E$ to engine 8. Both scenarios reduce the message hop cost, with a slightly higher steady state cost in the latter case.

The basic redeployment algorithm extended with the set redeployment optimization can find a better solution as show in Figure 6.14(c). Here, after moving activities $D$ and $E$ to the same engine, the algorithm moves both these activities together until they reach engine 1. The steady state message hop cost is now only 10.8% that of the initial deployment, and between 15.6% and 11.4% of the various runs of the basic redeployment algorithm.
6.4.5 Monitoring

In this experiment, we now add an SLA monitoring workflow to monitor the end to end execution time of the process. Recall that this workflow is itself mapped to a set of distributed activities, and can have a cost function applied to it. In this case, we wish to minimize message overhead of the monitoring workflow.

Fig. 6.15 shows the message overhead of only the monitoring workflow in the cases where a cost function is or is not applied to it. In both cases, the process being monitored is allowed to dynamically redeploy itself to an optimal configuration. We see in Fig. 6.15(a) that even though the monitoring workflow deployment is static, it’s overhead varies as the underlying process adapts itself. In Fig. 6.15(b), by continuously redeploying the monitoring process to reflect the current workload and deployment of the process, we achieve a message overhead of about 45% of the static monitoring deployment.

6.4.6 Sensitivity

This experiment evaluates how the redeployment algorithm reacts to a short but significant disturbance. The initial deployment of activities is that shown in Figure 6.8 and the initial conditions are $p = 0.9$ and $q = 0.1$. The primarily difference is that there is a short period of five seconds during which $q$ is increased to 0.999.
We see in Figures 6.16(a) and 6.16(b) that the basic redeployment algorithm reacts to this change; namely, it moves activity $E$ to engine 7 to minimize the cost of executing the tight loop between activities $E$ and $F$. However, after the burst is over, and $q$ returns to 0.1, we see that the message hop cost is now larger than it was before the burst. Not shown is that the system would eventually return to the previous deployment.

When the basic redeployment algorithm is extended with sensitivity analysis from Section 6.2.3, we see in Figures 6.16(c) and 6.16(d) that the system no longer reacts to the small perturbation in the workflow control flow patterns. In this way, the system avoids unnecessary activity redeployments and the system remains in a lower cost deployment after the temporary deviation in workflow characteristics.
6.4.7 Larger workflow

This experiment evaluates how a larger workflow consisting of fifty nine activities, and whose control flow is depicted in Figure 6.17, is managed. The activities here are initially deployed randomly across the engines in the system.

From the results in Figure 6.18(b), we see that that the dynamic redeployment makes a number of activity movement decisions over time that contribute to reducing the message cost of the workflow. The vertical lines that indicate when redeployments occur show that most of the activity movements take place early in the process run, and the movement frequency steadily decreases over time. After a little over 150 seconds, the
system stabilizes and achieves a message hop cost that is only 14% of the static random deployment in Figure 6.18(a).

### 6.4.8 Energy use

In this experiment, we use a cost function that seeks to minimize energy use. We assume that servers use a constant amount of energy, but unused execution engines (i.e., those with no activities deployed on them), can be turned off. To model this, the energy use of an execution engine is a step function with non-zero energy use when it hosts more than one activity.

Fig. 6.19 shows the number of execution engines with non-zero deployed activities when the process in Fig. 6.17 is deployed with a cost function that attempts to minimize energy use. We see that in the dynamic case activities are relocated to use only only one third the number of execution engines as the initial deployment, thereby reducing the total energy use of the system.
Chapter 7

Service and resource discovery

This chapter develops a distributed service discovery protocol that supports services with both static and dynamic attributes, and provides both a one-time and continuous query model. Section 7.1 describes the event-based resource discovery framework, and optimizations based on exploiting similarities among concurrent resource requests are outlined in Section 7.2. Finally, Section 7.3 presents and analyzes experimental results.

7.1 Resource discovery framework

This chapter proposes a new resource discovery framework based on the publish/subscribe model. The framework supports two types of resources (static and dynamic) and two types of discovery requests (one-time and continuous) resulting in four models as summarized in Table 7.1. The static and dynamic resource types distinguish between resources whose attributes are constant or may change over time. On the other hand, the one-time and continuous discovery request types denote cases where requests are matched against existing resources, versus ones where requests are also continually matched against newly registered resources. The algorithms for each of the four models are described in detail in this section.

What is common across the models is that resource providers act as publishers and
resource discovery clients act as subscribers. In order to allow a single system to contain resources and requests conforming to the different models, messages (including advertisements, subscriptions, and publications) are marked with a ModelType that specifies the model to be used. Valid ModelType values for discovery request messages are static, dynamic, static continuous (continuous query for static information), and dynamic continuous (continuous query for dynamic information). These values denote the kind of information the requester wants. For the resource, the ModelType can only be static or dynamic, indicating whether the resource’s attributes are all constant or whether some may vary dynamically.

The resource attributes and discovery constraints are specified as conjunctions of predicate constraints where each predicate is an [attribute,operator,value] tuple in which the operator and value specify a Boolean condition on the attribute. The attribute is a string, and predicate values may be integers, floating point numbers or strings. String types only support the equality operator, whereas the numeric types additionally support inequality operators ($<$, $\leq$, $>$, $\geq$, $=$).

For example, a resource with the static description “a Linux server with 2GB of memory and a 320GB disk” is represented by an advertisement “[system,=,Linux], [memory,¡=,2], [disk,¡=,320]” in the publish/subscribe system. Similarly, a discovery request for “Linux servers that have a disk larger than 120GB” is mapped to a subscription “[system,=,Linux], [disk,¿,120]”. As well, for dynamic resources whose attributes may vary frequently, their resource updates are conveyed with a publication. For example, a resource that currently has 1G of available memory and 200G of storage space could issue a publication such as “[system,=,Linux], [memory,1], [disk,200]”.

In some cases, a resource may only be available during certain time periods. Such
temporal availability constraints can be specified as predicates in both resource advertise-
ments and discovery subscriptions. For example, a resource that is only available after
8 a.m. can include the following predicate in the advertisement: “[available, \text{\&}, 8 \text{ a.m.}]”.\footnote{The time is specified in a convenient format here, but can be encoded using some mapping to an integer or floating point representation.} Similarly, a discovery request for resources available between 7 a.m. and 3 p.m. can add the following predicates to the subscription: “[available, \text{\&}, 7 \text{ a.m.}], [available, \text{\&}, 3 \text{ p.m.}]”.

### 7.1.1 Static model

The static model is designed to primarily handle information about static resources. The static resource description is registered with an advertisement message. This message is flooded and thus each broker caches information about all the static resources in the system. Recall from Section 2.1.2 that advertisements are cached in the subscription routing table (SRT) in the PADRES system. To discover static resources, the discovery client simply submits a subscription message containing the desired resource constraints to a broker. Upon receiving the subscription, the broker matches the subscription with the local advertisements and returns the resulting advertisements as a package back to the discovery client. Notice that the discovery request for static resources is handled by a single broker.

Figure 7.1 illustrates an example of resource discovery in the static model. In a network composed of brokers $A$, $B$, $C$, and $D$, a resource connected to Broker $A$ wants to register its resource information (ModelType=static, system=Linux, memory\text{\&}=2, disk\text{\&}=320). An advertisement $adv_1$ is generated that corresponds to these attributes and is flooded over the network. A discovery client connected to Broker $C$ submits its request for servers with at least 1GB of memory as subscription $sub_1$. On receiving the subscription, Broker $C$ queries its SRT for matching advertisements and then delivers these matches to the client. The subscription needs not be routed to the other brokers but is processed.
entirely by the broker the discovery client is connected to.

### 7.1.2 Dynamic model

There are cases where some attributes of a resource description vary over time, such as the available memory or processor utilization of a server. The dynamic model is proposed to support such resources.

In the dynamic model, each broker maintains a *PubCache* structure to cache updates to dynamic resource attributes. The *PubCache* consists of pairs $\langle advID, pub_i \rangle$, where $advID$ is the advertisement associated with a resource and $pub$ is a publication that contains the latest information about the dynamic attributes of the corresponding resource.

To register a resource in the dynamic model, a resource provider connects to a broker and issues a resource description advertisement (with $ModelType$ set to *dynamic*) that describes the ranges of its dynamic attributes. This advertisement is flooded across the network as in the static model. When resource attributes change, however, a publication message with the current resource attributes is generated and cached in the *PubCache* of the connecting broker.

To discover resources with dynamic attributes, the discovery client issues a subscrip-
tion message to its broker (with the ModelType set to dynamic). The subscription is routed to brokers along the matching advertisement trees towards those brokers where potentially matching resources are connected. These brokers are referred to as edge brokers. When the subscription reaches these edge brokers, the broker reads the latest information of the resource from the PubCache and routes the information back to the subscriber along the path the subscription just traveled.

Figure 7.2 shows an example of resource discovery in the dynamic model. A resource at Broker A registers its resource (ModelType=dynamic, system=Linux, memory\(i\)=2, disk\(i\)=320) with advertisement adv\(_i\). When the resource attributes change, the new values, say (system=Linux, memory=1, disk=200), are conveyed by issuing publication pub\(_i\) to Broker A which then caches pub\(_i\) in its PubCache. When a discovery client, connected to Broker C, wants to find servers with available disk space greater than 40GB, it issues subscription sub\(_i\). Since adv\(_i\) matches sub\(_i\), sub\(_i\) is routed along path C−B−A. At the last hop, Broker A reads the latest attributes of resource adv\(_i\) from its PubCache, which is pub\(_i\) in Figure 4, and routes pub\(_i\) back to the discovery client along the path A−B−C created by sub\(_i\).

Notice that the algorithm does not directly send discovery results for resources hosted at Broker A to the discovery client at Broker C. This makes it possible to take advantage of the similarities among resources and requests and share these results among concurrent discovery requests, thereby reducing network traffic and message processing loads. This optimization is developed and described in Section 7.2.

### 7.1.3 Continuous model

In both the static and dynamic models above, the discovery request will only find resources that were registered before the request was issued. In some scenarios, a discovery client may wish to be notified of any changes to the resources in the system, including whether resources matching some criteria are added or removed from the system, as well
The continuous model supports the ability of a client to submit a discovery request once and have matching resource information be delivered continuously as resources are added to the system or resource attributes are updated. As will be described, this model takes full advantage of the efficient matching and instantaneous message delivery capabilities of content-based publish/subscribe systems.

The continuous model is divided into two cases: a static continuous model to discover static resources, and a dynamic continuous model that is used to discover resources with dynamic attributes. The algorithms for both of these cases are outlined below.

**Static continuous model**

Discovery requests in the static continuous model are handled in a manner similar to static discovery requests as described in Section 7.1.1. The `ModelType` of the resource advertisements are still `static`, whereas the `ModelType` of the subscription request is now `static continuous`.

As in the static model, a broker that receives a discovery request subscription from a discovery client will query its SRT (which contains all the advertisements in the system), and return the matching resource advertisements to the client.
tinuous model, however, the broker will also store the discovery request in its routing tables. When the advertisement associated with any subsequent resource registrations are received, the broker will match the new advertisement with previously issued static continuous discovery requests and notify the client of these new matching resources.

In this way, a client need only submit its discovery request once, and will be notified of matching resources as they are registered in real-time. A client that is no longer interested in these notifications can submit a corresponding unsubscription message and the broker will remove the indicated discovery request from its tables.

**Dynamic continuous model**

The dynamic continuous model fully exploits the original advertisement-based publish/subscribe routing algorithms. To begin, the resource registers its resource description with an advertisement message with the ModelType attribute set to dynamic continuous that is flooded over the network. Next, a discovery client that wants to continuously monitor some resources issues a subscription message containing the desired constraints which is then routed to brokers with matching resources. Finally, when the dynamic attributes of a resource changes, the updated publication message is generated and is routed to the interested discovery request subscribers according to the routing algorithm in publish/subscribe systems.

Figure 7.3 illustrates an example of resource discovery in the dynamic continuous model. The resource connected to Broker A registers its resource description (system=Linux, memory=2, disk=320) with advertisement $adv_1$. Then a discovery client connects to Broker C as a subscriber, and indicates its interest in monitoring the status of all “Linux” machines (system=Linux) with subscription $sub_1$. This subscription is routed along path $C - B - A$ by tracing the reverse path of $adv_1$. Once the resource information changes, a new publication message is generated and routed back to the discovery client along the path $A - B - C$ traversed by $sub_1$. 
7.1.4 Discussion

To summarize, the ModelType of the advertisements that register resources can be static or dynamic, representing the appropriate characteristics of the resource. Discovery request subscriptions, on the other hand, can have ModelType values of static or static continuous when the client wants to find static resources, or values dynamic or dynamic continuous to find dynamic resources.

Note that the discovery for static resources is fully handled by a single broker and these subscriptions do not need to be routed through the network. This is similar to some of the centralized resource discovery approaches, although here there are a set of brokers that can each independently service requests for static resources, allowing the system to scale better than purely centralized schemes.

Even in the case of discovery requests for dynamic resources, however, request subscriptions are only routed towards brokers with potentially matching resources. Notably, if no advertisements match the request subscription, the subscription is not forwarded.

This chapter is mainly concerned with developing scalable resource discovery algorithms that support the models outlined above. While fault-tolerance is out of the scope of this work, it is worth pointing out that the system can continue to operate despite faulty clients. The system need not service a failed subscriber that has issued a discovery request, and the subscriber’s subscriptions can simply expire after some time. Similarly, the failure of publishers is handled by expiring its resource registrations after some time. In terms of broker failures, research on reliable publish/subscribe systems can be adopted to address broker fault-tolerance concerns [19, 41, 79].

7.2 Similarity forwarding

As described in Section 7.1, in the dynamic model resource discovery requests are routed to brokers which cache the dynamic attributes of matching resources. When the frequency
of discovery requests becomes high, the routing and processing of discovery subscriptions and update publications across the network degrade the performance of the system as a whole.

One strategy to optimize this cost is based on the assumption that in a given system, some number of concurrent discovery requests may be similar. For example, most resource discovery requests on PlanetLab may simply want to find machines with sufficient memory or storage resources: “find a machine whose available storage is more than 40GB” \( (req_1 = \text{storage} > 40) \), or “find a machine whose available storage is more than 50GB” \( (req_2 = \text{storage} > 50) \). In this case, the relationship between the two requests is that \( req_1 \) covers \( req_2 \), that is, the set of resources that match \( req_1 \)'s criteria is a superset of those that match \( req_2 \).

Consider a scenario where \( req_2 \) is issued shortly after \( req_1 \). In this case, it would be desirable to reuse \( req_1 \)'s results by filtering out those resources that do not match \( req_2 \) and route the remaining results to the client that issued \( req_2 \). Doing so avoids the need to route \( req_2 \) to all brokers with potentially matching resources and process the request at these brokers. The rest of this section outlines algorithms to find and exploit such similar discovery requests.
7.2.1 Definition of discovery similarity

It is useful to define a metric that quantifies the degree of similarity among discovery requests. Among other uses, this metric is applied in Section 7.3 to help measure the effects of request similarity on discovery performance. Since the approach in this chapter maps discovery requests to subscriptions, it suffices to define a similarity metric among a set of subscriptions.

Consider the constellation in Figure 7.4 where each dot represents a subscription, and some subscriptions are grouped into blocks. Each block represents a connected covering network, in which any subscription in it will cover or be covered by some other subscription in the same block. Isolated subscriptions have no covering relationships with other subscriptions and do not belong to a block.

It can be costly to compute and consider all the covering relationships among a large set of subscriptions. As the primary use for the metric in this chapter is to serve as an evaluation parameter, a relatively easy to compute metric that loosely captures the degree of similarity is sufficient.

**Definition 1**: The similarity of a set of subscriptions is a measure of the number of subscriptions with covering relationships compared to the total number of subscriptions.

Formally, subscription similarity is calculated as:

\[
\text{Similarity} = \sqrt{\frac{\sum_{i=1}^{b} a_i^2}{n}}
\]

where \( n \) is the total number of subscriptions in the set, \( b \) is the number of blocks, and \( a_i \) is the number of subscriptions in block \( i \). For the example in Figure 7.4, the parameters are \( n=30, b=6, \) and \( a=\{4, 5, 4, 3, 4, 7\} \), yielding a similarity of approximately 0.38 after substitution into the formula.

One way to think about the similarity formula above is that given a set \( S \) of \( n \) subscriptions with \( b \) blocks, the numerator in the expression is trying to find a single equivalent block of size \( a_e \) that represents the similarity of the subscriptions in \( S \). It is
clear that if $S$ only has one block of size $a_1$, the equivalent block should also have size $a_1$, and this is indeed the case in the above expression. When $b > 1$, however, it is less obvious what the equivalent block size should be, but it should be constrained by two bounds.

First, the equivalent block should be larger than any of the $b > 1$ blocks in $S$. Suppose this is not the case, and there is a block $i$ such that $a_i > a_e$. This would mean that another subscription set $S'$ also with $n$ subscriptions but with only one block of size $a_i$ would have a larger similarity metric than that of $S$: $\text{Similarity}(S') = a_i/n > \text{Similarity}(S) = a_e/n$. This is undesirable since the subscriptions in $S$ clearly have more covering relationships than those in $S'$ and the similarity metric should reflect this.

Second, the equivalent block should be smaller than the sum of the block sizes in $S$. Suppose this is not the case, and $a_e \geq a_m = \sum_i a_i$. Therefore a subscription set $S''$ with $n$ subscriptions but where the $b$ blocks in $S$ are merged into one block of size $a_m$ would have a smaller similarity measure than $S$: $\text{Similarity}(S'') = a_m/n > \text{Similarity}(S) = a_e/n$. This is undesirable since the subscriptions in $S''$ have more covering relationships than those in $S$ and again the similarity metric should reflect this.

By taking the square root of the sum of squares of each $a_i$, the similarity metric satisfies both these bounds.\footnote{By the Pythagorean Theorem, in a right triangle, $a^2 = b^2 + c^2$, where $a$ is larger than both $b$ and $c$, but smaller than $(b + c)$.} This justifies the similarity metric above.
7.2.2 Similarity forwarding algorithm

As described earlier, a client discovers resources by submitting a subscription message to a publish/subscribe broker. Covering relationships may exist among different discovery subscriptions, and this section outlines these relationships that are used to optimize the discovery cost.

In the discussion below, the broker that a discovery client connects to is referred to as the discovery host broker (DHBroker for short), and the broker that a resource connects to is the resource host broker (RHBroker). Also, the broker or client from which a resource registration advertisement, discovery request subscription, or discovery result publication is received is referred to as the last hop of the associated message.

As part of the similarity forwarding algorithm, each broker maintains three additional data structures: The $SrdSubList$ structure caches the discovery requests that are not covered by other requests in the current broker. The $ResourceCacheMap$ structure caches the discovery results for those requests for which the broker is a host broker, that is, requests from clients directly connected to the broker. Finally, the $SubWaitingListMap$ structure caches discovery requests waiting for the results from other discovery requests. The requests in this structure are covered by the ones in the $SrdSubList$ structure.

When no covering discovery requests have been seen by a broker, requests are forwarded as usual towards resource host brokers with potentially matching resources, which then return information about the matching resources to the requesting client. However, when a covering request is found, the new covered request retrieves the results directly from the discovery host broker where the covering request results have been cached. In addition, to account for the case where the results for the covering request have not been delivered to the requesting client yet, the covered request is also forwarded to brokers that may potentially have outstanding results.

In the similarity forwarding algorithm, when propagating a subscription $s$, the first broker $B$ that finds a subscription $s'$ in its routing table that covers $s$ forwards $s$ towards
### Algorithm 4: Subscription forwarding

**Input:** sub ← an incoming subscription message

1. $\text{matchingAdvs} \leftarrow \text{findMatchingAdvs}(\text{sub})$

if $\text{sub}.\text{payload} = \text{null}$ then

1. $\text{coverSub} \leftarrow \text{null}$;

// Look for a covering sub.

4. for $\text{srdSub} \in \text{SrdSubList}$ do

5. if $\text{srdSub}.\text{covers}(\text{sub})$ then

6. $\text{coverSub} \leftarrow \text{srdSub}$;

7. break;

8. end

9. end

10. if $\text{coverSub} = \text{null}$ then

11. $\text{SrdSubList}.\text{insert}(\text{sub})$;

12. route($\text{sub}, \text{lastHops}(\text{matchingAdvs}))$;

else

14. $\text{coverSub}.\text{waitingList}.\text{insert}(\text{sub})$;

15. if $\text{this.broker} = \text{coverSub}.\text{DHBroker}$ then

16. route($\text{coverSub}.\text{cachedResults}, \text{sub}.\text{lasthop}$);

17. else if $\text{coverSub}.\text{hasResults}()$ or $\text{coverSub}.\text{lasthop} \in \text{lastHops}(\text{matchingAdvs})$ then

18. $\text{sub}.\text{payload} \leftarrow \text{coverSub}$;

19. route($\text{sub}, \text{coverSub}.\text{lasthop}$);

20. end

21. end

22. else

23. $\text{coverSub} \leftarrow \text{sub}.\text{payload}$;

24. if $\text{this.broker} = \text{coverSub}.\text{DHBroker}$ then

25. $\text{coverSub}.\text{waitingList}.\text{insert}(\text{sub})$;

26. route($\text{coverSub}.\text{cachedResults}, \text{sub}.\text{lasthop}$);

27. else

28. if $\exists \text{neighbor} \in \text{lastHops}(\text{matchingAdvs}) : \text{neighbor} \notin \{\text{coverSub}.\text{lasthop}, \text{sub}.\text{lasthop}\}$ then

29. $\text{coverSub}.\text{waitingList}.\text{insert}(\text{sub})$;

30. end

31. if $\text{coverSub}.\text{hasResults}()$ or $\text{coverSub}.\text{lasthop} \in \text{lastHops}(\text{matchingAdvs})$ then

32. route($\text{sub}, \text{coverSub}.\text{lasthop}$);

33. end

34. end

35. end
the $DHBroker$ of $s'$. This is done in order to retrieve any cached results of $s'$ in the $DHBroker$’s $ResourceCacheMap$. Broker $B$ also stores $s$ in its $SubWaitingListMap$ so it can intercept any new results for $s'$. If a broker does not find any covering subscription, it propagates $s$ as usual based on the matching advertisements. Subscription forwarding in the similarity forwarding algorithm is detailed in Algorithm 4.

**Algorithm 5:** Publication forwarding

<table>
<thead>
<tr>
<th>Input:</th>
<th>$pub$ ← an incoming publication message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if $pub$.lastHop.isClient() then</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>cache($pub$);</td>
</tr>
<tr>
<td>3 else</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$s1$ ← $pub$.relatedSubMessage;</td>
</tr>
<tr>
<td>5</td>
<td>route($pub$, $SubWaitingListMap$.get($s1$));</td>
</tr>
<tr>
<td>6</td>
<td>route($pub$, $s1$.lastHop);</td>
</tr>
<tr>
<td>7 if $SrdSubList$.contains($s1$) and $s1$.lastHop.isClient() then</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$ResourceCacheMap$.insert($s1$, $pub$);</td>
</tr>
<tr>
<td>9 end</td>
<td></td>
</tr>
<tr>
<td>10 end</td>
<td></td>
</tr>
</tbody>
</table>

Resource update publications are cached at the resource’s $RHBroker$, and forwarded hop by hop towards any matching subscription’s $DHBroker$ where it is also cached. As well, each broker consults its $SubWaitingListMap$ to find any discovery requests that are waiting to intercept the update publication, and also propagates the publication towards these covered subscriptions. Publication forwarding is outlined in Algorithm 5.

The similarity forwarding algorithm caches some information at the brokers including the discovery requests and discovery results. These cache entries expire after some globally configured expiration time (such as 100s). Longer expiration times offer more opportunities to exploit similarity among discovery requests at the expense of returning possibly stale results.

### 7.2.3 Similarity forwarding example

Figure 7.5 presents an example of the similarity forwarding algorithm. The example consists of a seven broker network (Brokers $A$ to $G$), two discovery requests ($D1$ and $D2$)
and three resources (R1, R2, and R3), where it is assumed that all three resources match both discovery requests, and that discovery request D1 covers D2.

In Figure 7.5, a client connects to Broker E, and submits its discovery request D1. Broker E finds no requests in its SrdSubList that cover D1 and so adds D1 to its SrdSubList. The broker then forwards D1 to neighbors from which it has received advertisements that match D1, Brokers D and G in this case. This process is repeated at each broker, including adding D1 to the SrdSubList if no covering requests are found, until D1 reaches all the resource host brokers with matching resources, Brokers A, F and G in this case. Next, the information about the matching resources R1, R2, and R3 are routed back to Broker E along paths A − C − D − E, F − D − E, and G − E, respectively. In addition, the information about resources R1, R2, and R3 are cached in Broker E’s ResourceCacheMap. The tables in Figure 7.5 summarize the key state at each broker in the network, including the fact that Brokers A, C, D, E, F, G have added request D1 to their SrdSubList. Note that in reality these tables are distributed among the brokers.
and are only presented together in the figure for convenience.

Continuing the scenario in Figure 7.5, suppose a new client connects to Broker B, and issues discovery request D2. Broker B finds no discovery requests in SrdSubList that cover D2, adds D2 to SrdSubList, and finally routes D2 to neighbors with matching advertisements, which is only Broker C in this case. Broker C, however, finds that there is a request, D1, in SrdSubList that covers D2. At this point Broker C begins the process of looking for results that match D1 in two places: at those brokers that may have already received these results, and those that may still receive new results.

For the former case, D2 is forwarded to D1’s last hop (which from Broker C’s point of view is Broker D) until it reaches D1’s host broker where any cached results are retrieved from the host broker’s ResourceCacheMap. For the latter case, Broker C stores D2 into its SubWaitingListMap in order to intercept new results for discovery request D1. The new results may arrive from neighbors other than the last hop of D2 (Broker B) and the last hop of D1 (Broker C). In this scenario, Broker A is the only such neighbor, which incidentally is also the host broker of R1. Similarly, D2 is inserted into the SubWaitingListMap at Brokers D and E.

At Broker E, the host broker of D1, the algorithm filters the cached results for D1 preserving only those resources that also match D2, and routes the results back along the path E − D − C − B. When the results finally arrive at Broker B, the algorithm also caches the results here to be used by any future discovery requests that are covered by D2.

In this way, the similarity forwarding algorithm exploits the concept of subscription covering in publish/subscribe systems to find discovery requests whose results may be shared. As well, distributed publish/subscribe routing algorithms are used, with some modifications, to ensure that requests are forwarded to all brokers that potentially have already or will receive shared results. In situations with many concurrent discovery requests for overlapping sets of resources, this algorithm can provide significant benefits,
as evaluations in Section 7.3 confirm.

7.3 Evaluation

The primary objective of this section is to observe, quantify, and understand the performance of the resource discovery algorithms presented in this chapter. Different workloads are evaluated in particular those with varying degrees of similarity among the discovery requests.

7.3.1 Setup

The algorithms in this chapter have been implemented in Java over the PADRES distributed content-based publish/subscribe system [29]. The experiments are conducted on a real deployment across a cluster of machines that represent a data-center environment. The network topology consists of 20 brokers each running on a cluster node, as well as 4 other brokers that are central in the network and simply act as publish/subscribe routers. In order to accurately measure some of the metrics, all the clients for issuing the experimental workloads run on a separate node.

The evaluations measure four metrics. The average discovery time represents the overall time duration from when a discovery request is issued to when the results are returned. The number of publication messages is the number of hops traversed by publications, and likewise for the number of subscription messages. Recall that subscriptions correspond to discovery requests and publications to results. Finally, the matching operations metric counts the number of times a matching operation is performed by the brokers. In the PADRES system, which internally uses the Jess rule engine to perform the matching operations [54], the number of matching operations is simply the number of times the Jess engine is invoked.
7.3.2 Parameters

To isolate against the effects of several parameters, the experiments start with a simplified, basic setting. It is assumed that no failures occur at either the resource or node level [37]. Also, all the resources are registered in the network before the discovery requests are issued and are not unregistered during the experiment. This avoids the influence of the fixed, known costs of resource registration from the discovery cost which is of more interest.

The details about the resources and discovery requests used in the experiments are outlined in the following sections.

Resources

So as to simplify the experiments, only five tags ($a$, $b$, $c$, $d$, $e$) are used to represent the resource attributes. Each resource randomly selects at least two attributes, and each attribute’s value is randomly selected from the range $1$ to $100$.

During the experiment, the dynamic information of these deployed resources change randomly, both in terms of the frequency of the updates, and the resource parameter value in the update, subject to being within the range that the resource advertised during registration. Recall that these updates are not propagated over the network but only refresh the cache at the resource’s host broker.

Resources are registered at brokers in the network according to one of two distributions: balanced and unbalanced. In the balanced distribution, resources are uniformly registered across the network, whereas in the unbalanced distribution, the resources are deployed following a geometric distribution $P(X = n) = (1 - p)^{n-1}p$, with $p = 1/3$. In the latter case, most of the resources are registered at a small number of brokers. In total 1000 resources are deployed across the 20 nodes. Figure 7.6(a) shows the number of resources deployed on each node for the two distributions.
Discovery requests

Ten discovery request workloads are generated, each containing 1000 discovery requests, and with each workload having similarities ranging from 10% to 100%. Each generated discovery request contains attribute \( a \) and some of the other four attributes \( b, c, d, e \). In order to remove the effect of different discovery result sizes on the query time, we take into account the average result size of discovery results in each workload. Figure 7.6(b) shows the average number of resources matched by a discovery request workload.

Among these generated discovery requests, some may match one or more resources, while others may match none. The former are referred to as valid discovery requests, and the latter invalid. The similarity of a workload is varied by replacing invalid discovery requests with ones that are covered by valid ones.

In the experiments, discovery requests are varied in both time and space. The time distribution refers to how the requests are issued over time, with the number of requests issued per unit of time following various Gaussian distributions as illustrated in Figure 7.7(a). The space distribution, on the other hand, is concerned with which brokers the requests are issued to, and in this case a Zipf distribution \([14]\), as shown in Figure 7.7(b), is used. This means that most requests will originate from a small set of
7.3.3 Results

Two sets of experiments are presented here. The first set evaluates the performance of the similarity forwarding algorithm under a variety of workloads, followed by a set of experiments that quantify the costs and benefits of a decentralized resource discovery architecture compared to a centralized one.

Similarity forwarding

This section evaluates the effect of similarity among discovery requests on the performance of the discovery algorithms under various environments.

For comparison, both the un-optimized algorithm presented in Section 7.1.2 and the optimized similarity forwarding algorithm are evaluated. The un-optimized algorithm is denoted Normal, and the optimized one Similarity.

Figures 7.8, 7.9, and 7.10 show the results of the discovery algorithms with different workloads. In each experiment, both balanced and unbalanced resource distributions are considered. Various discovery request distributions are presented as follows: balanced in
both time and space (Figure 7.8); balanced distribution in time and Zipf distribution in space (Figure 7.9); and Gaussian distribution in time and balanced distribution in space (Figure 7.10). In each figure, there is a chart for each of the four metrics: the overall discovery time, the publication message traffic, the subscription message traffic, and the number of times a matching operation is executed.

Overall, the results show that the similarity forwarding optimization can significantly reduce the overall discovery execution time, network traffic cost, and processing overhead. Furthermore, the benefits grow when the discovery requests exhibit increasing similarity, demonstrating the ability of the similarity forwarding algorithm to exploit workloads where the results among discovery requests can be shared.

In terms of the publication messages (the second chart in Figures 7.8, 7.9, and 7.10),
all the experiments show that the similarity forwarding algorithm greatly reduces the number of publications. It does this by, when possible, retrieving results from the caches at discovery request host brokers instead of collecting the results from each individual resource host broker.

One seemingly odd result is that the number of publication messages (which are used to deliver discovery results) grows with increasing similarity. However, this is expected because in the workload the number of valid discoveries (which match at least one resource) increases with the similarity degree, thereby resulting in an increase in the number of matching resources, and hence more publication messages, as the discovery requests become more similar.

The number of subscription messages (the third chart in Figures 7.8, 7.9, and 7.10) remains relatively constant for the normal un-optimized discovery algorithm. Recall that subscriptions, which correspond to discovery requests, are routed towards potentially matching resources according to their registration advertisements. Since the average number of matching advertisements for each workload is relatively stable, as seen in Figure 7.6(b), it is not surprising that the number of subscription messages is also stable. With the optimized similarity forwarding algorithm, however, subscriptions sometimes only need to be routed to the host broker of a covering subscription rather than to all host brokers of all the matching resources. These savings increase when more similarity is available to be exploited.

The processing time spent by brokers executing matching operations (the fourth chart in Figures 7.8, 7.9, and 7.10) closely tracks the number of subscriptions (the second chart in the figures) for both algorithms. This is a straightforward result of each broker attempting to match subscriptions against the advertisements in its routing tables. However, with the similarity algorithm there are cases where a subscription that is covered by another one can simply be forwarded toward a particular host broker without having to first find matching advertisements. The results do indeed confirm that the matching cost
with the similarity forwarding algorithm can be less than the number of subscriptions especially when there is more similarity among the requests, and more temporal locality of similar requests (Figure 7.10).

There is an interesting effect of the spatial distribution of the resources in the network. In terms of the average discovery latency, messages and matching time, both the normal and similarity forwarding algorithms perform better when the resources are deployed according to the unbalanced rather than balanced distribution in Figure 7.6(a). The reason for this is that in the unbalanced distribution, most resources tend to be located in one part of the network and the discovery messages are more likely to be isolated to this part of the network.

**Decentralized architecture**

This section compares the decentralized resource discovery architecture proposed in this chapter with a centralized deployment. All four models discussed in Section 4 are evaluated to measure the time it takes to find and report the matching resources for a set of discovery requests.

The decentralized deployment is the 24 broker network used in the earlier experiments, whereas the centralized deployment consists of a single broker to which all clients connect. The workloads for each model are as follows.

In the static model, 1000 resources are first registered and then 1000 discovery requests are issued. In the dynamic model, 1000 resources and requests are issued as in the static model, but this time the attributes of the resources are updated frequently throughout the experiment. In the static continuous model, 20 discovery requests are issued, one to each broker in the decentralized case but all to one broker in the centralized case. Then, 1000 resources are registered. In the dynamic continuous model, 20 resources are registered followed by 20 discovery requests distributed as in the static continuous case. Then, the attributes of the 20 resources are updated throughout the experiment.
In all the above cases, the resources and discovery requests follow the uniform spatial distribution for the decentralized deployments. For the two dynamic cases, each resource generates 50 resource update publications for a total of 1000 publications.

Figure 7.11 shows the results of evaluating the four models. In Figures 7.11(a) and 7.11(b), sets of discovery requests with increasing number of expected resource matches are issued. Each data point represents the average query time of the requests in the corresponding set of requests. The results show that unlike in the centralized architecture, the decentralized deployment, by effectively distributing the load, maintains a relatively constant overall query time despite an increasing number of matching results. The continuous model results in Figures 7.11(c) and 7.11(d) also show that overall discovery time is better when there are multiple brokers in the system.

The results in Figure 7.11 show that the multiple brokers in the decentralized architecture can deliver discovery results faster than the centralized resource discovery architecture in which the single broker can become a bottleneck. The tradeoff, however,
is that the decentralized architecture imposes additional network traffic to propagate the discovery requests and results among the broker network.

Figure 7.12 quantifies this decentralization cost by plotting the total number of message hops for the experiments in Figure 7.11. While the message overhead may be large, it is important to note that despite these additional messages, the earlier results showed that the system as a whole is able to perform the discoveries faster and offer a more responsive and scalable service to the discovery clients.

Another set of experiments were conducted to evaluate whether it is better to send discovery results directly back to the requesting client without traversing through the brokers in the overlay. This algorithm, which was briefly mentioned at the end of Section 7.1.2, is denoted Direct and is compared to the Normal unoptimized discovery algorithm. The results under a balanced resource and discovery request distribution and 50% similarity are shown in Figure 7.13.

The experiments show that directly sending results to the requesting client can significantly reduce the publication traffic but provides negligible benefits when considering the other metrics, notably the overall discovery time. The Direct method also suffers from requiring a new connection to be established between every pair of matching resource host broker and discovery request host broker, something that may not be feasible in
some environments for performance or security reasons. As well, by sending the results back, the algorithm makes it impossible to use the similarity forwarding techniques to share results among similar discovery requests.

The earlier experiments show that the similarity forwarding algorithm significantly improves all four metrics (overall discovery time, publication and subscription traffic, and matching time). Although the publication traffic costs are better with the Direct method, on balance the similarity forwarding algorithm may be the better choice.
Chapter 8

Automatic service composition

The chapter develops an algorithm to create a user-specified service composition by searching through a distributed set of service registries. The reader is encouraged to first refer back to Section 2.3.2 for some of the terminology used in this chapter. Section 8.1 presents a distributed process search architecture including the mapping of the problem to publish/subscribe terms and an algorithm to realize distributed, parallel process search. This is followed with an evaluation of the algorithm in Section 8.2.

8.1 Distributed service composition

This chapter presents a distributed architecture for automatic service composition consisting of service agents and request agents connected to a distributed content-based publish/subscribe broker network as illustrated in Figure 8.1. Services are assigned to service agents which register the service by translating service input and output interfaces into publish/subscribe advertisement and subscription messages. Request agents, on the other hand, manage the requests and results of process searches. Both service and request agents connect to the broker network as ordinary publish/subscribe clients and do not need to be concerned with the internal details of the publish/subscribe matching and routing algorithms.
The system and algorithm are fully distributed with no centralized data structures or control nodes. Service and request agents can connect to arbitrary brokers, and brokers and agents can be added or removed at any time. Furthermore, the decoupling properties of the publish/subscribe network allow the agents to coordinate a distributed process search among themselves without being aware of one another.

Section 8.1.1 describes how the broker network indirectly establishes the compatibilities among registered services, and Section 8.1.2 presents a distributed algorithm whereby the service agents collaborate to discover service compositions that satisfy a user’s search request.

### 8.1.1 Mapping to publish/subscribe model

This section outlines how the service composition problem can be expressed in terms of content-based publish/subscribe messages.

Recall that a service $W$ is defined by its input and output interfaces, $W_{in} = \{I_1, \ldots, I_m\}$ and $W_{out} = \{O_1, \ldots, O_n\}$, respectively. The input interface is mapped to a set of subscriptions and the output interface into an advertisement, such that a service is now defined as $W = \{W_{subs}; W_{adv}\}$. 
Chapter 8. Automatic service composition

![Diagram of services and requests]

Figure 8.2: Example services and requests

<table>
<thead>
<tr>
<th>Service or request</th>
<th>Subscriptions</th>
<th>Advertisement</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimplePrinter</td>
<td>$s_0 : (\text{postscript} = &quot;\ast\ast\ast&quot;)$</td>
<td>$a_0 : (\text{print_status} &gt;= 0)$</td>
</tr>
<tr>
<td>DuplexPrinter</td>
<td>$s_1 : (\text{postscript} = &quot;\ast\ast\ast&quot;)$</td>
<td>$a_1 : (\text{print_status} &gt;= 0)$</td>
</tr>
<tr>
<td></td>
<td>$s_2 : (\text{duplex_option} &gt;= 0)$</td>
<td></td>
</tr>
<tr>
<td>HTMLConv</td>
<td>$s_3 : (\text{html} = &quot;\ast\ast\ast&quot;)$</td>
<td>$a_3 : (\text{postscript} = &quot;\ast\ast\ast&quot;, \text{pdf} = &quot;\ast\ast\ast&quot;)$</td>
</tr>
<tr>
<td>WordConv</td>
<td>$s_4 : (\text{word} = &quot;\ast\ast\ast&quot;)$</td>
<td>$a_4 : (\text{postscript} = &quot;\ast\ast\ast&quot;)$</td>
</tr>
<tr>
<td>Request1</td>
<td>$s_5 : (\text{print_status} &gt;= 0)$</td>
<td>$a_5 : (\text{postscript} = &quot;\ast\ast\ast&quot;)$</td>
</tr>
<tr>
<td>Request3</td>
<td>$s_6 : (\text{print_status} &gt;= 0)$</td>
<td>$a_6 : (\text{html} = &quot;\ast\ast\ast&quot;, \text{duplex_option} = 0)$</td>
</tr>
</tbody>
</table>

Table 8.1: Subscriptions and advertisements for services in Figure 8.2

$W_{subs}$ is a set of $m$ subscriptions, each of which contains a single predicate corresponding to a parameter in the input interface: $W_{subs} = \{s_1, \ldots, s_m\}$ (1 ≤ $i$ ≤ $m$) where $s_i$ corresponds to $I_i$. The representation of a parameter by a subscription will depend on the expressiveness of the subscription language. For a predicate-based language that supports string comparisons as described in Section 2.3.2, the subscription may simply specify a globally unique string representation of the parameter type. For a WSDL Web service, this may be the URI associated with the input message schema definition of an operation. For a Java RMI method, this may be the fully qualified class name of the method’s argument types.

Notice that the input interface is mapped to a set of subscriptions each with one
predicate instead of a single subscription with a conjunction of predicates. This is required so that the input to a service can be matched by the outputs of a set of services instead of a single service.

The output interface of service $W$ is mapped to a single advertisement $W_{adv}$ with $n$ predicates, each corresponding to a parameter in the output interface: $W_{adv} = \{O_1, \ldots, O_n\}$. As with the subscriptions, the representation of an output interface parameter as predicates in an advertisement will depend on the advertisement language supported by the publish/subscribe system.

For an invocation $R = \{R_{in}, R_{out}\}$, the input is mapped to an advertisement, and the expected output to a subscription in a similar manner to the service interfaces: $R = \{R_{adv}, R_{subs}\}$, where $R_{subs} = \{s_1, \ldots, s_q\}$.

Table 8.1 lists the subscriptions and advertisements issued for each service in Figure 8.2. Notice that the two input parameters of service $\text{DuplexPrinter}$ are mapped to two subscriptions with one predicate each, but the two output parameters of the $\text{HTMLConv}$ service are represented by one advertisement with predicates.

The sections below continue the mapping of the problem by redefining the terms from Section 2.3.2 in terms of publish/subscribe primitives.

**Successor**

Service $W'$ succeeds service $W$ if every subscription in $W'_{subs}$ intersects $W_{adv}$: $\forall i \in \{1, \ldots, m\}, s_i \triangledown W_{adv}$, where $\triangledown$ is the intersection operator between subscriptions and advertisements.

For example, in Table 8.1, $\text{SimplePrinter}$ succeeds $\text{WordConv}$ because every subscription by $\text{SimplePrinter}$ ($s_0$ in this case) intersects $\text{WordConv}$’s advertisement $a_4$. 
Compatibility

Service $W$ is compatible with service $W'$ if at least one subscription in $W_{subs}$ intersects $W_{adv}$: $\exists i \in \{1, \ldots, m\}, s_i \triangleright W_{adv}$.

From Table 8.1, DuplexPrinter is compatible with HTMLConv, because the former contains at least one subscription ($s_1$) that intersects the latter’s advertisement ($a_3$).

Atomic Process

As before, a service $W$ fully satisfies an invocation $R$, iff $R_{adv}$ precedes $W_{subs}$, and $W_{adv}$ precedes $R_{subs}$.

For the services in Table 8.1, service SimplePrinter satisfies invocation Request1 since Request1 precedes SimplePrinter ($a_5 \triangleright s_0$), and SimplePrinter precedes Request1 ($a_0 \triangleright s_5$).

Composite Process

An invocation $R$ is fulfilled by a composite process represented by a DAG $D$ consisting of a sequence of services $\{W_1, W_2, \ldots, W_x\} (x \geq 2)$ iff

- $R_{adv}$ precedes $W_{1subs}$;

- For every subscription $s \in W_{isubs} (1 \leq i \leq x)$, there is an advertisement $a \in \{R_{adv}, W_{1adv}, \ldots, W_{i-1adv}\}$ where $a \triangleright s$; and

- For every subscription $s \in R_{subs}$, there is an advertisement $a \in (R_{adv}, W_{1adv}, \ldots, W_{xadv})$ where $a \triangleright s$.

It can be shown that following the above rules will result in the same DAG results as in Figure 2.4.
8.1.2 Process search algorithm

As described earlier, a service agent registers services in the system by issuing appropriate advertisement and subscription messages into a publish/subscribe broker (network). This section outlines how the relationships maintained by the publish/subscribe system among subscriptions and advertisements is exploited to discover a composition of services that satisfy an invocation’s input and output requirements.

Before a search for processes fulfilling invocation $R$, it is assumed that every service $W$ is associated with a service agent, and that the agent has issued the corresponding $W_{adv}$ and $W_{subs}$ messages, and as part of the initialization of the search, the search request agent has issued $R_{adv}$ and $R_{subs}$.

Recall that by issuing the above messages, the publish/subscribe system routes the advertisements and subscriptions in such a way the intersection relationships among them form a directed (possibly) cyclic graph $G$. The objective of the process search algorithm is to discover DAGs within $G$ that satisfy the atomic or composite process properties defined in Section 8.1.1.

At a high level, the search algorithm works by injecting a publication into the system that is successively delivered to compatible service agents. At each step, a DAG is built by appending a compatible service to the DAG until a DAG is found that originates and terminates at the request agent.

- The request agent issues a publication $p$, where every predicate constraint in advertisement $R_{adv}$ occurs as an attribute-value pair in $p$. The publication also includes a representation of a DAG $D$ as a payload. This DAG is successively built as the publication propagates, as described below. At this point, the DAG consists of the single node $R$.

- Publication $p$ is delivered to all service agents with compatible services. An agent for service $W$ creates a new DAG $D'$ by appending $W$ to the DAG in $p$, constructs
a new publication $p'$ that corresponds to its advertisement $W_{adv}$, includes $D'$ in $p'$, and injects $p'$ into the publish/subscribe system.

- If a process fulfilling invocation $R$ is found, a publication $p$ from services compatible with $R_{subs}$ is delivered to the request agent, which appends $R$ to the DAG in $p$. The resulting DAG describes a process that fulfills the requirements in $R$, and the DAG is then returned to the user that requested the search.

The algorithm described above omits certain details of the service agent functionality. In particular, the process DAG should consider the successor relation among neighboring services, not compatibility relations. The service agent is responsible for aggregating compatibility relations into suitable successor relations as described below.

### 8.1.3 Service agent

A service agent represents a service, and is responsible for registering the service with the system, and participating in the process search algorithm. A service agent for service $W$ consists of the following components.

- **Publish/Subscribe client**: This component performs publish/subscribe messaging, including issuing publications, subscriptions, and advertisements, and receiving matching publications.

- **Publication cache**: Publications from services that are compatible with $W$ but do not precede $W$ are stored in a cache until a set of publications is accumulated that together precede $W$. This is described in more detail below.

- **Successor matching**: This component executes the algorithm in Figure 6, and is responsible for finding services in the publication cache that precede $W$. 
As outlined above, a service agent waits until it receives a set of publications \( P_i \) from services \( W_i \) that cumulatively precede the local service \( W \). The algorithm to determine this is given in Figure 6.

**Algorithm 6:** Incrementally search for a process

**Input:** \( pub \) ← a publication with a process search request

1. \( pubCache \leftarrow pubCache \cup pub \);
2. \( md \leftarrow \emptyset \);
3. **if** for each \( s \in W_{subs} : s \) matches \( pub \) **then**
   4. \( dag \leftarrow pub.payload \);
   5. \( md \leftarrow dag.append(W) \);
5. **else if** there exists a minimal set \( P \subseteq pubCache \) such that \( \forall s \in W_{subs} : \exists p \in P \) that matches \( s \) **then**
   7. **forall** the \( p \in P \) do
     8. \( dag \leftarrow p.payload \);
     9. \( dag \leftarrow dag.append(W) \);
   10. \( md \leftarrow md.merge(dag) \);
8. **end**
6. **end**
13. **if** \( md \neq \emptyset \) **then**
14. \( p \leftarrow generatePub(W_{adv}, pub) \);
15. \( p.payload \leftarrow md \);
16. \( send(p) \);
17. **end**

If a single service \( W' \) precedes \( W \), \( W \) is simply appended to \( W' \) in the DAG, representing a *sequence* relationship between \( W' \) and \( W \).

On the other hand, when a set of publications \( P_i \) are required to precede \( W \), this represents an *and-join* relationship between the services \( W_i \) and \( W \). Therefore, there must be a corresponding *split* relationship in the DAG. The *merge* function in the algorithm in Figure 6 creates these *split* points by merging the largest common prefixes of the DAGs in \( P_i \). The *merge* function can be implemented by a modified topological sort algorithm.

Not shown in Figure 6 is that the publications cached by the service agent expire after some preset time. The publications cannot be removed after a match is found for them because they may be used as part of another result for the same search request. For example, suppose that as part of the search, DuplexPrinter’s service agent in Figure 2.3
receives publications from Request3 and HTMLConv, finds a match, publishes the resulting DAG (in this case the first one in Figure 2.4), and discards the two publications from its cache. Then when it receives a search publication from PDFConv it would no longer discover the second DAG result in Figure 2.4 because it has discarded the publication from Request3.

8.1.4 Discussion

Some additional details about the distributed automatic service composition algorithm deserve some attention.

Deadlocks

It is possible that there is no composition of services that fulfills an invocation request, that is there are no results. Because the search algorithm is distributed, it is difficult to determine when the search has completed.

A process search with no results has two implications: the requester waits indefinitely, and cached publications at service agents are never flushed. A simple way to resolve these problems is to expire the cached publications after some time, and for the request agent to wait for some bounded time for results. The timeout periods must consider the potential complexity of results (processes that are a composition of many services will typically take longer to find), the distribution of service agents (communication among agents imposes an overhead), and the complexity of the interface representations (interfaces represented by parameter type names can probably be matched quicker by the distributed publish/subscribe broker network than complex XML schema definitions).

Note that even in the case where results are found for a process search request, the distributed nature of the search algorithm may still leave unused publications in the publication caches in service agents. This is because not all branches of the search tree reach back to the request agent. For example, consider the compatibility graph in
Figure 8.3: Dead search paths

Figure 8.3 where, for simplicity, it is assumed that every service has exactly one input and output parameter thereby making each compatibility relation a successor relation as well. In this example, a result for the invocation is found as $R_{in} \rightarrow W_2 \rightarrow W_4 \rightarrow W_6 \rightarrow R_{out}$. However, there was also a search path $R_{in} \rightarrow W_1 \rightarrow W_3$ that did not terminate at $R_{out}$. The publications along these dead paths need to be expired.

**Livelock**

The graph of service compatibility relationships may have cycles, and therefore, the search algorithm may traverse these cycles. For example, for the compatibility graph in Figure 8.4, an infinite number of DAGs can be found, with zero or more instances of $W_4$. To avoid this case, a condition is added to the successor matching algorithm in Figure 6 so that the service agent for service $W$ drops publications whose payload contains a DAG that already includes $W$.

Figure 8.4: Composition with loop
Concurrent searches

There may be multiple requests for a process search, from one or more request agents, occurring simultaneously in the system. To distinguish these searches, the request agent includes a unique request identifier to the publication it sends to trigger the search, and all publications sent by service agents include this identifier. The algorithm in Figure 6 only considers sets of publications with the same identifier when determining predecessor relations. In this way, concurrent searches do not affect one another.

Reuse search results

It is possible that processes found as a result of a search may fulfill another search in its entirety or as a part of it. To facilitate these cases, the request agents, can register the resulting processes of a search for invocation \( R = \{R_{in}, R_{out}\} \), as another service \( W = \{R_{in}, R_{out}\} \). In this case, \( W \) is a composite service, and the service agent will append the entire composite process (as opposed to a single service) to the DAG as part of the successor matching algorithm.

Process constraints

Process search can be restricted to find processes with constraints such as the maximum depth or maximum number of service compositions. The former can be implemented by using a time-to-live (TTL) field in the search publications that is decremented at every service agent. When the TTL field reaches zero, the publication is discarded, and the search terminates along that path. Another approach is for the service agents to count the number of composed services in the DAG included in the search publication payloads. DAGs with more than the desired services are discarded.

Other constrains include restrictions on the monetary cost of a process (assuming each service imposes some price to invoke it), security restrictions (perhaps processes should not include services from certain combinations of service provides), parallelism (processes
with wide split and join nodes require more computation by process execution engines to collect and aggregate results from services executing in parallel), or search time (a search can be terminated if it is taking longer than the requester is willing to wait for it). Such constraints will be explored in more detail in future extensions to this work.

Including such constraints in the distributed search algorithm prunes unnecessary search trees early and avoids delivering invalid results to the requester. This benefits the user and reduces the overhead of the search on the system.

### 8.2 Evaluation

This section experimentally evaluates the automatic service composition algorithm presented in this chapter under various scenarios and workloads in order to determine the strengths and weaknesses of the algorithm.

![Figure 8.6: Search result structures](image)

(a) DAG structure  
(b) Chain structure
8.2.1 Setup

The distributed process search algorithm has been implemented over a distributed content-based publish/subscribe system [29]. The experiments are run in a 14 node cluster of 1.86 GHz machines each with 4 GB of RAM. At most one publish/subscribe broker is deployed on each machine, and one or more service agents are connected to each broker.

Three deployments, described below, are evaluated.

- **Centralized**: One publish/subscribe broker is deployed on one machine, with one service agent connected to the broker. This deployment attempts to simulate a centralized process search architecture.

- **Distributed**: One broker is deployed on each of the 14 machines, for a total of 14 brokers forming the overlay shown in Figure 8.5. As well, 14 service agents are deployed, one per machine, and connected to their local broker. This deployment represents a truly distributed environment and process search algorithm.

- **Hybrid**: One broker is deployed on one machine, and 14 service agents are deployed, one per machine, and connected to the single broker in the system. This deployment, when compared to the above two, is used to isolate the performance impact on a process search by the publish/subscribe brokers and service agents.

In all cases above, one request agent is deployed on one machine and connected to the local broker. Unless otherwise specified, 250 services are randomly assigned to the available service agents.

The processes that are results of searches are either a complex DAG shown in Figure 8.6(a), or a chain of services of variable length illustrated in Figure 8.6(b). These two process structures are used to study effects such as parallelism, and control factors such as the process length.

The metrics of interest are the search latency and message overhead. The search latency is measured as the duration from when a process search is issued by a request
agent to when the agent receives a response. In the results, the latency is typically averaged over a number of search requests. The message overhead counts every hop that every publication traverses in the overlay network, and is normalized to the number of search requests issued to ease comparisons across different experiments. The focus is on publication messages because advertisements and subscriptions are only issued during service registration and are not propagated as part of the search algorithm. The latency measure is probably of more importance to the user, while the message count may be interesting to an administrator of the system.

8.2.2 Results

The experiments are grouped in terms of the parameter that is varied, and the results are analyzed in detail for each set of experiments.

Search frequency

This experiment studies the impact of the number of concurrent process search requests. Each request results in a single result that looks like the process in Figure 8.6(a). Two hundred requests are issued in total, but the interval between successive requests is varied from 10 ms to 1000 ms.
Figure 8.7(a) plots the average search latency (on a log scale) for the various request intervals and shows that when the request rate is very low (i.e., large request intervals) the centralized and hybrid deployments outperform the distributed one, and all of the searches take approximately 200 ms to 250 ms. As the request rates are increased, however, the performance of the centralized and hybrid schemes degrade significantly, whereas the distributed deployment remains relatively stable. This is because as the request rates increases, the number of concurrent searches increase as well, and the distributed scheme is able to process the searches in parallel.

It is interesting to note in Figure 8.7(a), that the sharp increase in search latency for the centralized and hybrid approaches occurs when the request interval is less than 250 ms, which is roughly the average time it takes to process a single request. In other words, when the request rate exceeds the rate of search evaluation, the system becomes overloaded, queuing delays increase, and search performance degrades drastically. The distributed scheme, however, by parallelizing the search processing, is able to maintain stable search performance even when the request rate exceeds this threshold of 250 ms, but does eventually become overloaded when the request interval is less than 10 ms. Incidentally, note that as there are 14 brokers, an optimal parallelization of the search should take about $250 \text{ ms} / 14 \text{ brokers} = 18 \text{ ms}$. Therefore it is expected that a request interval of 10 ms will overwhelm the system.

Figure 8.7(a) shows that by distributing the service agents (but not the brokers), the hybrid deployment achieves slightly better search latency than the centralized case, but the similarity of their results indicates that the matching done at the publish/subscribe brokers dominates the processing at the service agents. The similarity between the hybrid and centralized deployments is consistent across all the evaluations.

The message overhead of the three deployments is presented in Figure 8.7(b). Comparing the three cases, the distributed approach experiences the highest message overhead compared to the centralized and hybrid schemes, which is an intrinsic tradeoff with any
distributed algorithm. Notice that there is almost no variance in the message overhead indicating, as expected, that the request interval has no effect on the steps taken by the search algorithm. This supports the argument above that the variations in search latency in Figure 8.7(a) are due to queuing delays (as the system becomes overloaded) and not because of increased matching time or parallelism effects.

Results per search

This experiment evaluates the effect of the number of search results per request. A single request is issued each time, but different number of processes are found to match this request. To isolate the effect of the number of results, each result is a simple chain of fourteen services of the structure shown in Figure 8.6(b).

Figure 8.8(a) presents the search latency for a varying number of results per search.
The latencies for all three approaches are linear with the number of search results. However, the distributed approach benefits from the parallelism opportunities when there are more results per search, and therefore is less sensitive to this parameter.

Notice in Figure 8.8(b) that although the distributed deployment suffers from the highest message overhead, because these messages are distributed across the available resources, the distributed scheme is still able to provide the lower search latencies in Figure 8.8(a).

Process length

The size of the processes found to satisfy a search has an impact on the performance. In this experiment, requests are issued sequentially (no parallelism), each request returning a process that is a simple chain structure (see Figure 8.6(b)) but with varying length, so that the number of services composed by the process differs.

The results in Figure 8.9(a) show that the search latency increases with the result length. Now, the distributed approach performs the worst, and the difference between the distributed and centralized schemes widens with longer paths. The reason for this is because the results in this experiment are chains which afford no opportunity for the distributed approach to parallelize the search; the resulting process is discovered sequentially one service at a time.

To investigate the effects of parallel searches when the results are chains, an additional experiment is run for the 70 process length case, but this time the requests are repeatedly issued with an interval of 1000 ms between each request. Note that this interval is less than the average single request latency of about 3200 ms for the centralized scheme when the process length is 70, so some searches will be processed in parallel. The results in Figure 8.9(c) compare the sequential and parallel search scenarios, and show that the distributed approach maintains a stable search latency whereas the hybrid and centralized schemes are extremely overloaded (note the log scale in Figure 8.9(c)).
The results in Figure 8.9(c) show that when results are long processes, the centralized scheme may outperform the distributed one when there are few concurrent searches, but is not the appropriate deployment choice if many simultaneous searches are expected.

Unlike the experiment from Figure 8.7(b), the message overhead in this experiment increases with the result length as shown in Figure 8.9(b). The almost identical trends in Figures 8.9(a) and 8.9(b) indicates that the latency increases are primarily due to having to match more messages both in the publish/subscribe system and the service agents. However, the impact of the service agents is minimal because the hybrid case benefits little from distributing the service agents, and therefore it can be inferred that the publish/subscribe matching at the brokers is the more significant factor.

Service deployment

In the distributed deployment, the location of services in the network may affect the search performance. This experiment repeats the case in Section 8.2.2 where five results are found for a search request, but varies the assignment of services to service agents.

Two extreme service deployment cases are considered: an in-order one in which services adjacent in the search result are deployed to service agents at adjacent brokers in the topology in Figure 8.5; and an alternating deployment where consecutive services in the search result chain are assigned to brokers at opposite ends of the network, specifically,
Chapter 8. Automatic service composition

Figure 8.11: Registered services

to the service agents connected to brokers 1 and 13 in the topology in Figure 8.5.

As well, two search strategies are used: a sequential one where only one request is issued at a time, and a parallel one where requests are issued with an interval of 1000 ms, a duration small enough to ensure some searches are processed in parallel.

Figure 8.10 shows that with sequential requests, the alternating deployment is about 47% worse than the in-order one, whereas the difference is about 390% when requests are issued in parallel. The results show that while both sequential and parallel searches suffer from a poor deployment of services, such as the alternating deployment, the parallel searches are more sensitive to the service distribution because the impact of the extra publications traversing back and forth between the ends of the network is amplified by the number of concurrent searches taking place.

Registered services

Each registered service injects a subscription and advertisement into the publish/subscribe broker network, and imposes additional state at its service agent. Even if a service is not found as part of a search, it may impact the performance of the search because of the matching overhead imposed by its subscriptions and advertisements on the publish/subscribe brokers. This experiment investigates the effect of “background” services that are registered in the system but are not composed as part of any process.
A single request is issued (no concurrent requests) with a resulting process of the structure in Figure 8.6(a). Figure 8.11 presents the average search latency for this request with varying number of “background” services randomly assigned to service agents. The results indicate a large impact from these additional services. The centralized approach clearly does not scale, with more than a 170 fold worse search latency when the number of services is increased from 100 to 900. The distributed deployment scales much better with a sublinear inflation of only about 265% for the 800% increase in the number of services.

**Service similarity**

It is desirable for a process search algorithm to exploit similarities between registered services to prune the search space or optimize the search. Fortunately, there has been much work in publish/subscribe matching research on exploiting *covering* among subscriptions and advertisements [16].

This experiment repeats the one from Section 8.2.2 with 900 additional “background” services and a single search result of the form in Figure 8.6(a). However, the similarity of these additional services is controlled. To achieve $x\%$ similarity, $x\%$ of the services are randomly chosen to be identical to one of five possible services, and the remaining services are mutually different with no common input or output parameters.
Figure 8.12 shows that in all three deployments, increasing similarity of services results in smaller search latencies. Results show that the message overhead does not vary with similarity, and therefore the latency savings are primarily due to the publish/subscribe matching algorithm’s ability to perform matching faster when there are more covering subscriptions, that is, more services with common interfaces.
Chapter 9

SLA modeling and monitoring

This chapter develops solutions to model and monitor Service Level Agreements (SLAs) that specify the guarantees negotiated between service providers and consumers. Section 9.1 describes the details of the SLA model, and Section 9.2 presents the SLA monitoring architecture. This is followed by a case study in Section 9.3 where the SLA model is applied to an example insurance claim application process.

9.1 SLA model

An overview of the solution in this chapter is illustrated in Figure 9.1. As shown in the figure, when both business process and SLA are complete, they are validated to ensure the SLA is still applicable to the business process. This is required because the established relationship might be broken during parallel development of the two artifacts. Subsequently, they are taken as inputs to the generation procedure.

It is assumed that during execution the business process execution engine is capable of emitting events containing a snapshot of the current state of the business process. It is also assumed that the business process can be modeled as a directed graph with control flows that describe the business logic. These assumptions are essential to the proposed SLA model which extends the concepts from WSLA [38] with some refinements. These
refinements simplify the existing model without losing any critical information in defining an SLA.

The SLA model is extensible and inheritable to allow maximum reuse. Complex SLAs can be constructed by composing a number of simpler SLAs and their components. Figure 9.2 shows the overall structure of the proposed model.

Each model has a reference to a business process being monitored. The reference is simply a file location of the business process definition, which can be expressed in
any supported process modeling language. The model also includes a reference to the
documentation detailing the contract on the level of service agreed by both parties. It is
usually used by SLA developers as a reference during SLA modeling.

SLAs are the topmost elements of the model. As shown in Figure 9.2, hierarchical
relationships are established between SLAs and other components in the model. Top
level components have dependency relationships with the components below. In addition,
components are reusable so that multiple components can establish a dependency with
the same component. In subsequent sections, each component is discussed in detail.

9.1.1 Scopes

A scope defines a region or subsection of the business process being monitored. The SLA
model allows multiple scopes to be defined. A scope is expressed as a series of start and
end node pairs in the process diagram. For example, Figure 9.3 shows a process diagram
in which each activity is identified by a unique id. The figure contains two sections that
overlap each other. The left region can be expressed as \( \{1, 4\} \) because it begins at node
1 and ends at node 4. This notation describes all path entries that begin with node 1
and end with node 4. Thus, paths 1-2-4 and 1-3-4 are considered to be in this region.

The proposed scope definition is flexible enough to express a wide variety of regions.
For example, to express the scope on the right in Figure 9.3, we can define as a series of
start and end node pairs, such as \( \{\{4, 6\}, \{4, 7\}\} \). The sequence captures all path entries
within this region. More complex regions can be expressed in a similar fashion.
9.1.2 Metrics

Metrics play an important role in the proposed SLA model. It defines an indicator for measuring different aspects of a process. Not only does a metric measure the performance of a process, but it also captures other information that is critical to the evaluation of the SLA. For example, the usage of a service by all customers might be measured to evaluate whether the subscribed service is underused. Once the measurements are taken by metrics, one can easily construct a Service Level Objective (SLO) by assembling the metrics together.

In the proposal, a metric is expressed as an instance of a metric type in a metric type library. A metric type library is a collection of reusable metric types which users can augment with their own metric types. A metric type defines the specific data and format to be captured as well as the methodology and business logic in order to measure this piece of information. In addition, it specifies the set of events that triggers a measurement to be taken. In general, a metric type consists of the following components:

**Type identifier**: A unique identifier is assigned to every metric type in the library. Because of its uniqueness, a new metric can be extended from a metric type by referencing the types identifier.

**Parameters**: A list of required parameters is defined for every metric type. Metric types make use of these parameters for their operation. The actual values of these parameters will be provided at a later time when a metric instance is defined. Detaching the parameter values from the schema gives more flexibility to users to customize the metrics. For example, a user can create multiple metrics extending from the same metric type. By assigning different parameter values to each metric, he is able to customize them to measure different aspect of the process.

Every parameter in the schema specifies its data type and optionally its range. Parameter types include scalars, scopes, individual activities and other metrics. The scope parameter can define the regions where the metric applies, thus measurement is only
taken for the given scope. Some metric types do not specify a scope as a parameter, as the target region is implicitly defined. For instance, the metric type for measuring the transaction time of the process does not require a scope, because the scope is implicitly defined to be the entire process.

Metrics can also be included as parameters of other metric types. These metric types usually require other metrics to assist in taking measurements. For example, to measure the percentage of successful execution of a process, a metric might require other metrics that measure the total number of executions and the total number of successful executions. As a result, the resulting metric type might require these two metrics as parameters.

Dependent events function: This function captures a list of events that must be emitted by the process. These events are critical because they trigger the dependent metrics to take measurements. For instance, to measure the process execution time, the corresponding metric type must be notified of the entry and exit events of the process. Thus, the dependent events function for this metric type is defined as \( \{A_{\text{entry}}, Z_{\text{exit}}\} \), where \( A \) is the start activity and \( Z \) is the end activity of the business process.

The function is invoked in various places in the solution. It is first called during validation to collect a set of events that must be emitted by the referenced process. Validation then ensures the process is capable of emitting all of the events as required by the SLA. It is also invoked to retrieve the list of events and enable them in the referenced business process.

Event handler: The event handler of a metric type contains the logic of how a measurement should be taken. It is executed when a relevant event, specified in the dependent events function, is emitted. The triggering event is passed as input to the handler. In general, it is a function call during runtime execution. The handler retrieves necessary data from the event for its processing.
Figure 9.4: Credit check process

<table>
<thead>
<tr>
<th>Type identifier</th>
<th>InvocationCountType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Scope (type: scope)</td>
</tr>
<tr>
<td>Dependent events function</td>
<td>{Start_entry}</td>
</tr>
<tr>
<td>Event handler</td>
<td>OnEvent(Event e) {</td>
</tr>
<tr>
<td></td>
<td>static total;</td>
</tr>
<tr>
<td></td>
<td>total = total + 1;</td>
</tr>
<tr>
<td></td>
<td>publish(total, e.instanceid);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Table 9.1: Metric type for measuring number of invocations

### 9.1.3 Metric type example

To illustrate the concept of metric types and metrics, an example is given in this section. In the loan approval example process, assume that credit check services are purchased from a third party. As depicted in Figure 9.4 the process checks whether the customers credit history is located in a local database and retrieves it locally. Otherwise, it retrieves the information from a remote database, which is a more expensive call. An SLA ensures the percentage of remote database accesses is under a certain threshold.

To measure the percentage of remote accesses, it is necessary to measure the number of invocations of both local and remote accesses. Thus, two metrics are required. However, because both metrics serve a similar purpose, they can actually be extended from the same metric type, which is shown in Table 9.1.

This metric type takes as a parameter the identifying the area of interest. In this scenario, the two activities retrieve data from the database. To measure the number of
invocations, the metric must be notified when the entry event of the scope occurs. When this happens, the event handler is invoked. The event handler updates the number of invocations and publishes an update event to notify others of the change.

Given the above metric type, the required metrics can thus be created to measure the number of invocations for both local and remote access. Tables 9.2 and 9.3 summarize the resulting metrics.

The next step requires the measurement of the percentage of remote access. Table 9.4 shows the metric type that calculates the percentage. The percentage metric takes the above two metrics as input parameters and is notified when any of the input metrics emit an update event to indicate a value change. When the percentage metric is notified, the event handler is invoked to compute the new percentage. The percentage metric then fires an update event to notify others who are interested in the state change.

Given the above metric type, a new metric can be created, as shown in Table 9.5. The

<table>
<thead>
<tr>
<th>Name</th>
<th>LocalAccessCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric type</td>
<td>InvocationCountType</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope = {LocalAccess}</td>
</tr>
</tbody>
</table>

Table 9.2: Metric for measuring number of local accesses

<table>
<thead>
<tr>
<th>Name</th>
<th>RemoteAccessCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric type</td>
<td>InvocationCountType</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope = {RemoteAccess}</td>
</tr>
</tbody>
</table>

Table 9.3: Metric for measuring number of remote accesses

<table>
<thead>
<tr>
<th>Type identifier</th>
<th>CalcPercentageType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>metric1 (type: InvocationCountType)</td>
</tr>
<tr>
<td></td>
<td>metric2 (type: InvocationCountType)</td>
</tr>
<tr>
<td>Dependent events function</td>
<td>{Update}_metric</td>
</tr>
</tbody>
</table>
| Event handler   | OnEvent(Event e) {
|                 | static percentage;
|                 | percentage = metric1 / (metric1 + metric2) * 100%
|                 | publish(percentage, e.instanceid); |

Table 9.4: Metric for computing percentage
9.1.4 Service level objectives

A Service Level Objective (SLO) defines a goal of the SLA. It is expressed as a Boolean expression in terms of metrics. The proposed SLA model allows multiple SLOs to be defined in one single model. Similar to metrics, SLOs are defined by extending an SLO type in the library. Continuing from the example in Section 9.1.3, in order to define the SLO that the percentage of remote database access over total number of access is under a threshold, one can make use of the SLO type in Table 9.6.

The above SLO type requires two parameters for evaluation: the metric that computes the percentage of remote accesses, and a scalar value that represents the desired threshold. If an SLO extends from the above SLO type, it will be triggered when an update event

<table>
<thead>
<tr>
<th>Name</th>
<th>CreditCheckSLO</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLO type</td>
<td>LessThanSLOType</td>
</tr>
<tr>
<td>Parameter</td>
<td>metric = RemoteAccessPercentage threshold = 10%</td>
</tr>
</tbody>
</table>

Table 9.7: Example of SLO
is emitted from the dependent metric, and its event handler is invoked to evaluate the condition. As shown in the table, a violation event is emitted when the condition is violated, so that interested parties are notified. This event captures the violated SLO which event recipients are able to retrieve. The resulting SLO is shown in Table 9.7.

9.1.5 Action handlers

As mentioned earlier, when an SLO is violated, it is desirable to take appropriate actions upon it. For example, it might require emailing a manager, or charging the service provider a penalty. The proposed model addresses this requirement by action handlers. Action handlers contain the logic that is executed upon a violation of an SLA. An action handle is typically a function written in any given programming language, and is invoked during process execution when a violation of an SLO is detected. To achieve a high degree of flexibility, the proposed model allows multiple action handlers to be defined within a single SLA model.

9.1.6 Service level agreements

As different aspects of an SLA are modeled in the proposed model, the SLA section is used to compose them together. An SLA is expressed as a set of pairs of SLOs and action handlers. The SLO specifies the goal of the SLA, whereas the action handler defines the action taken if the SLO is violated. To continue with the loan approval example, in order to model the SLA that ensures the percentage of remote access is under a predefined threshold, the SLA can be expressed as:

\[
SLA = \{\text{PercentageSLO}, \text{chargeProvider}\}
\]

Occasionally, several action handlers are required to execute upon a SLO violation. On the other hand, violations of multiple SLOs might cause the same action handler to run. For these reasons, the definition of SLA can be further generalized to encourage maximum reuse of SLOs and action handlers:
\[
SLA = \{\{SLO_1, \ldots, SLO_n\}, \{Action_1, \ldots, Action_n\}\}
\]

### 9.1.7 Discussion

The proposed model poses a number of advantages in modeling SLAs for any given process. First of all, the model itself defines metrics by extending metric types. By detaching the metric implementation as metric types, our model encourages reuse of metrics. As some metrics are commonly used by different processes, this architecture provides a great benefit to SLA developers.

Secondly, metric types make use of scopes and parameters to allow users to customize metrics to fit their needs. For example, by defining a metric type that measures the time taken for a scope, users are able to reuse it to measure different regions within a business process. Multiple metric instances can be created that extend the same metric type, each of which is then assigned to different scopes of interest. Another example can also be found in Section 9.1.3, in which two metrics extend the same metric type to measure the number of invocation for different scopes. In addition, users can even further customize metrics by overwriting the default implementation of event handlers. It thus provides the capability to fine tune any metrics in the model.

Furthermore, the architecture also encourages complex metrics to be constructed by composing other metrics. Section 9.1.3 presented an example where a metric type consumes other metrics to compute the percentage of remote access over total access.

Similarly, the model architecture also encourages reuse of SLOs and action handlers. As an SLO extends an SLO type, SLA developers can reuse the same type to create multiple SLOs. In addition, once an SLA is created, multiple SLAs can set it as their goal, which maximizes reusability of events further. Action handlers can also reused by multiple SLAs in the same fashion.

Because of the high reusability of the proposed model, it becomes easy to apply a SLA model from one process to another. For instance, to apply an SLA to another business
process, one can update the business process definition to refer to the new process. As discussed in Section 9.2, a validation algorithm ensures that the SLA is applicable to the new process. It is possible that the scope definition is no longer valid, or the process itself is unable to emit events that are critical to the SLA execution. Any of these incompatibility issues will be indicated by validation, which requires a user to correct them.

9.1.8 Implementation

The proposed SLA model has been implemented, and an editor allows users to create and modify the proposed model. The editor is developed and integrated into WebSphere Integration Developer, a screenshot of which is shown in Figure 9.5.

Users are able to use the editor to develop an SLA model by creating metrics and SLOs from a type in the corresponding libraries. In addition, users are also able to create new
metric and SLO types in these libraries through an extension framework, which provides the extensibility and flexibility capability introduced in the proposed model.

9.2 SLA monitoring

The previous section discussed the details of the proposed SLA model. In this section, the architecture to monitor the SLA is presented. As a first step, the SLA and its referencing process must be validated to ensure that the business process satisfies all requirements of the SLA; for example, all SLA scopes are resolved in the process.

After validation, the SLA and its referencing process are passed to the generation tool which generates the monitoring artifacts. It first derives the set of critical events used by the SLA and enables these events in the business process. The generation then generates a set of monitoring artifacts to execute the SLA. These artifacts respond to the events emitted by the process and thus evaluate the SLA.

To execute the generated monitoring artifacts, they must be deployed to a monitor server. Similarly, the referencing business process is deployed onto a process server. During the execution of the process, the monitoring artifacts monitor the process and ensure the SLA is not violated. In the following sections, details of the entire process will be discussed.

9.2.1 Validation

The validation process involves multiple steps. First, all scopes are validated to ensure they can be resolved in the process. In other words, the start and end activity of any scope must exist in the referencing process. Secondly, it also ensures that metrics are assigned with the correct set of parameters, as defined by their metric type.

Validation also analyzes the SLA and the business process to ensure the process is able to emit the set of events that are critical to SLA execution. The process begins
Chapter 9. SLA Modeling and Monitoring

by discovering all SLOs defined in the model. Starting from the SLOs, the validation process retrieves its dependent metrics from its hierarchical relationship. The process continues recursively until all nested metrics are obtained. For each metric, validation invokes its dependent events function to discover the set of critical events. The union of all resulting event sets is the set of events critical to the SLA execution. Given this event set, validation then ensures each event in the set can be emitted by the business process.

9.2.2 Event enabling

As discussed earlier in this paper, the architecture listens to events emitted from the business process. It invokes the event handler of the affected SLAs to evaluate if the current process state violates the SLAs. Typically not all events are necessary for SLA monitoring. For example, an SLA to ensure the process execution time is under a limit requires only two events to be emitted. They are the entry and the exit event of the referenced process. In platforms where it is costly to emit events, the performance overhead can be reduced by enabling only the set of the events that are critical to the SLA evaluation.

The same algorithm used during validation is used to determine the set of critical events. In summary, all SLOs defined in the model are discovered, the discovered SLOs are used to discover the dependent metrics, and the process continues recursively until all elementary metrics are found. After that, the dependent events function is invoked to discover the set of critical events. The union of all resulting event sets is the set of events critical to the SLA execution. Given this event set, the generation tool enables them in the referenced process.

9.2.3 SLA execution

In addition to event enabling, the generation tool also generates a set of monitoring artifacts for the given SLA. These artifacts, when executed, monitor the business process
As discussed in Section 9.1, the proposed SLA model is structured hierarchically. Top level components depend on components below. If the state of any child element is changed, it potentially has an impact on the elements above in the model. For example, in the credit check process example, when an entry event of the local access activity is emitted, the corresponding metric must be notified to update the number of invocation. As a result of the update, other metrics that depend on this metric must also be updated. In this case, the metric that computes the percentage must be updated. As shown in Figure 9.6, there is ripple effect that causes a reevaluation of the SLO. Notice that a publish/subscribe pattern is followed, and as a result, it is natural to disseminate these events over a publish/subscribe platform.

Figure 9.6 shows the runtime architecture that executes SLAs. In the system, SLA components such as metrics, SLOs and action handlers become the clients of the publish/subscribe system. They all act as both a subscriber and publisher of events. To determine the set of events a client is interested in, its dependent events function is called to retrieve the list and subscribe to the relevant events.
To execute the SLA, the target business process is first registered as a publisher of the system. During execution, the process emits events previously enabled by the generation.

As shown in Figure 9.7, when events from the business process enter the system (1), the publish/subscribe middleware forwards the events to interested clients. If the client is a metric, its event handler is invoked to update its value. The handler also emits new events into the system because of the update. These events are in turn forwarded to other interested clients (2), causing other clients in the system to reevaluate (3). This chain of actions continues until it reaches the top level clients (4). In this case, the action handler is executed because the corresponding SLO is violated.

This architecture poses a number of advantages. First, all clients in the publish/subscribe system are loosely coupled, which encourages reuse of these clients during runtime execution. For example, if several SLAs depend on same set of metrics, the monitoring server can reduce overhead by reusing its metric instances. It can easily be accomplished by switching program context when executing a metric.

Furthermore, loose coupling also encourages SLA execution in a distributed environment. Since the architecture is highly modular, distributed computing can be easily achieved by moving these clients around the server cluster.

### 9.3 Case study

In order to illustrate the proposed solution of modeling SLAs for a given business process, an end-to-end scenario is presented in this section. In this example, an insurance company
models its claim approval process as shown in Figure 9.8. The claim approval process is composed of a number of activities, and begins when an applicant submits a claim form. An automated process first evaluates the application, and the application is automatically processed if enough information is provided and the applicants credit history is good. Otherwise, the application is passed to a junior clerk to manually examine. Depending on the complexity of the application, she might escalate the request to her supervisor to handle. No matter who processes the application, an evaluation is given as a result. If the application is approved, the claim process continues with issuing a cheque to the applicant. Otherwise, the process replies to the customer with the reason for the rejection.

The insurance company is interested in enforcing several SLAs to govern the given claim process. The company wants the total time taken by the manual evaluation to be less than 200 hours per day. In addition, because supervisors are expensive resources, the percentage of the time spent by all supervisors must be less than 20 percent of the
<table>
<thead>
<tr>
<th>Type identifier</th>
<th>TotalExecTimeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Scope (type: scope)</td>
</tr>
<tr>
<td>Dependent events function</td>
<td>{Start_{entry}, End_{exit}}</td>
</tr>
</tbody>
</table>
| Event handler   | OnEvent(Event e) {
|                 |     static total = \{i_1, \ldots, i_n\};
|                 |     static entry = \{i_1, \ldots, i_n\};
|                 |     if (e is entry_event) {
|                 |         entry[e.instanceid] = e.time
|                 |     } else {
|                 |         diff = e.time - entry[e.instanceid];
|                 |         total[e.date] = total[e.date] + diff;
|                 |         publish(total[e.date], e.instanceid);
|                 |     }
|                 | } |

Table 9.8: Metric type for measuring execution time

total time taken by all persons. Also, the senior management team must be notified if any SLA is violated.

### 9.3.1 Modeling the SLAs

Given the above requirements, an SLA model can be created. To begin with, the SLA model first references to the claim approval business process. If a document is available that describes the SLA details, a reference to this document can be added in the SLA model.

#### Metrics

To continue modeling the required SLAs, the SLA developer must first identify the required metrics. Looking at the SLA requirements, it is known that three metrics are needed. The first metric measures the total execution time taken by all individuals, the second metric measures the total execution time taken by all supervisors only, and the last metric calculates the percentage by composing the first two metrics.

To define the first and second metrics, it is noticed that both metrics have similar nature in which both measure the execution time of a region of the process. As a result,
Table 9.9: Metric for measuring total execution time

<table>
<thead>
<tr>
<th>Name</th>
<th>TotalTimeByAllPersons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric type</td>
<td>TotalExecTimeType</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope = { {EvaluationByClerk}, {EvaluationBySupervisor} }</td>
</tr>
</tbody>
</table>

Table 9.10: Metric for measuring execution time of supervisors

<table>
<thead>
<tr>
<th>Name</th>
<th>TotalTimeByAllSupervisors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric type</td>
<td>TotalExecTimeType</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope = { EvaluationBySupervisor }</td>
</tr>
</tbody>
</table>

it is thus preferable to define a metric type to measure the total execution time, as shown in Table 9.8.

The metric type measures the total execution time required by a given scope. It listens to the entry event of first activity of scope and the exit event of the last activity of scope to record the total execution time of the region. When an entry event is received, the event handler records the start time of the region. Similarly, the handler records the finish time when an exit event is received.

As discussed in this paper, it is assumed that the event captures a snapshot of the current state of the process; thus the start and finish time can be retrieved from the event itself. Once both times are recorded, the total execution time can be calculated and the result can be published to interested components. Given the above metric type, one can define the required two metrics as follows:

Table 9.9 shows the first metric which calculates the total execution time spent by all individuals. This metric extends from the metric type that is defined previously. Two scopes are assigned as input parameters when the metric is declared, so as to achieve the objective of measuring the total execution time spent by all individuals, which include junior clerks and supervisors. Similarly, Table 9.10 extends the same metric, but is assigned one single scope which enables measuring the total execution time taken by all supervisors.

In addition to the above metrics, another metric is needed to calculate the percentage
of the time spent by all supervisors over all individuals. The metric type from Table 9.4 can be reused for this purpose, and instantiated as in Table 9.11. It takes the above two metrics to calculate the percentage, and its value is updated by the event handler when an update event is emitted by any of the two metrics. If its value is updated, the event handler in turn emits an update event to notify interested parties.

### Service level objectives

After all required metrics are defined, the SLA developer is able to declare the necessary SLOs. In this scenario, two SLOs are needed for the two SLAs being modeled: one to ensure the total execution time is less than a threshold, and one to check that the resulting percentage is less than a threshold. As a result, the SLO type from Table 9.6 can be used to model both SLOs, and instantiated as in Tables 9.12 and 9.13.

<table>
<thead>
<tr>
<th>Name</th>
<th>PercentageTimeBySupervisors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric type</td>
<td>CalcPercentageType</td>
</tr>
<tr>
<td>Parameter</td>
<td>( \text{metric1} = \text{TotalTimeByAllSupervisors} ) ( \text{metric2} = \text{TotalTimeByAllPersons} )</td>
</tr>
</tbody>
</table>

Table 9.11: Metric for computing percentage

<table>
<thead>
<tr>
<th>Name</th>
<th>TotalTimeByAllPersonsSLO</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLO type</td>
<td>LessThanSLOTType</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope = { \text{TotalTimeByAllPersons} } \text{Threshold} = 100</td>
</tr>
</tbody>
</table>

Table 9.12: Example of execution time SLO

<table>
<thead>
<tr>
<th>Name</th>
<th>PercentageUnderTwentySLO</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLO type</td>
<td>LessThanSLOTType</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope = { PercentageTimeBySupervisors } \text{Threshold} = 20</td>
</tr>
</tbody>
</table>

Table 9.13: Example of percentage SLO
Chapter 9. SLA modeling and monitoring

Figure 9.9: Implementation of action handler

Service Level Agreement

As mentioned earlier in the section, it is required the senior management team is notified if any SLA is violated. To accomplish the task, an action handler is written to inform the management team through e-mails. In this scenario, the action handler is implemented as in Figure 9.9.

The input of the handler is the violation event. It captures the current state of the business process. For example, it captures the SLO that is violated such that the handler is able to take appropriate action.

Once the action handler is implemented, the SLA developer can model the required SLAs as follows: The first SLA ensures that the total execution time used by all individuals is less than 100 hours per day, whereas the second SLA determines if the percentage of time spent by supervisors is less than 20 percent. If either SLA is violated, the management team will be notified by the action handler.

\[
SLA_1 = \{TotalTimeByAllPersonsSLO, sendNotify\}
\]

\[
SLA_2 = \{PercentageUnderTwentySLO, sendNotify\}
\]

9.3.2 Validation

To execute the SLAs above, a validation must be first executed to ensure the SLAs are applicable to the business process. First, all scopes defined in the model are validated. The start and end activity of every scope must exist in the process. Second, all metrics are extended from a metric type and they are assigned with the correct number of parameters.
As the final step, the validation retrieves a list of events that are critical to the SLA execution. It then ensures the process is capable of emitting these events. As discussed earlier, the validation algorithm transverses the hierarchy of the model starting from the SLOs. A list of SLOs and metrics that are used in the model can thus be obtained. For each element, the dependent events function is invoked to retrieve the set of critical events. In this scenario, the claim process must be able to publish the entry and exit event of the supervisor human activity, as well as the entry and exit event of the clerk human activity.

9.3.3 Generation of monitor rules

After validation, the SLA and its business process are given to generate the set of monitoring artifacts. That is, the set of clients that will be registered in a publish/subscribe infrastructure.

The generation first retrieves the list of critical events and enables them in the business process. As a result, the business process, when executed, emits the required events to the infrastructure. Next, the generation generates a monitoring client for each metric, SLO and action handler in the model. In this scenario, a total of six clients are registered in the system, three of which are generated from the metrics, two of which from the SLOs and one from the action handler. These clients are then registered into the system. During registration, each client also subscribes the set of events as defined by its dependent events function.
Chapter 10

Conclusions

This chapter summarizes the major contributions and results of this thesis, and presents potential directions to further this research.

10.1 Summary

This thesis addressed a number of problems regarding the management of business processes or composite applications. In particular, it developed solutions to support an architecture that is distributed and offers more flexibility in executing, monitoring, and discovering services. These goals are in line with the increasingly distributed nature of business process environments, and the need for business to be flexible and adapt to dynamic environments and business opportunities.

The solutions are built over a distributed content-based publish/subscribe overlay network. The publish/subscribe overlay offers a simple yet powerful interface that decouples the interaction among the application components. Motivated by some of the requirements of a distributed business process execution platform, however, the publish/subscribe overlay is extended with support for more dynamic client interactions. Chapter 3 adds support for the guaranteed movement of publish/subscribe clients according to well-defined properties. The transactional concerns of various layers are outlined, and
atomicity, consistency, and isolation properties for three layers—client, notification, and routing—are specified. Furthermore, protocols to achieve these properties are described and correctness proofs are given. Unlike protocols where client mobility is handled by the end point brokers, a more efficient reconfiguration algorithm is developed in which brokers on the path from the source to target brokers participate in the client mobility protocol. Evaluations indicate that the proposed reconfiguration mobility protocol outperforms the traditional protocol in terms of both the movement transaction time and network overhead, and confirm that the covering optimization is costly in a scenario with mobile clients. Moreover, the reconfiguration protocol exhibits much more stable behaviour: changes in the nature of the subscriptions, number of moving clients, and background publish/subscribe activity, such as unsubscriptions by non-mobile clients, hardly affect the performance of the reconfiguration protocol, whereas the traditional mobility protocol’s performance varies greatly. This stability property not only illustrates the scalability of the protocol, but is also vital for administrators to plan the provisioning of a network’s resources without having to be concerned with changing application workloads.

Chapter 4 addresses a neglected and perhaps largely unknown problem of severe congestion triggered by the removal of subscriptions in a content-based routing network. In particular, the popular covering optimization which aggregates a set of subscriptions with a single subscription \( S \), is susceptible to subscription bursts when \( S \) is unsubscribed. We propose a light-weight and fully distributed congestion control algorithm. Instead of tearing down an entire subscription propagation tree when an unsubscription is issued, the algorithm incrementally prunes portions of the tree. The pruning decisions are made locally by each broker which considers both the congestion that would be triggered by removing the aggregate covering subscription and forwarding the replacement covered subscriptions, as well as the false positive publications that would arrive if the subscription were not removed. Real quantitative comparisons of implementations of the proposed incremental covering algorithm with the traditional ones reveal severe congestion under
the traditional algorithms that are virtually eliminated with the incremental algorithm. In addition to sharply reducing the network congestion, the proposed algorithm can provide much smaller routing tables and thereby faster broker matching operations. These results are evident in every experiment we conducted including ones using a set of real traces of the distributed business process execution engine developed in this thesis. We conclude from the results that the traditional covering algorithms are not suitable for applications that may experience subscription churn, and in particular a large number of unsubscriptions. The congestion that results in such scenarios is excessive, and persist for an extended period of time. For example, in some experiments, queue lengths in the thousands of messages hardly diminished even after almost two hours. The proposed incremental algorithm, on the other hand, experienced virtually no congestion.

Next, we build a distributed business process execution engine over a content-based publish/subscribe overlay extended with the above capabilities. In Chapter 5 we propose a distributed business process execution architecture, based on a publish/subscribe infrastructure, using light-weight activity agents to coordinate the business process execution in a distributed environment. The publish/subscribe layer simplifies the interaction among agents, and reduces the cost of maintaining execution state for running process instances. Second, we describe how BPEL activities can be mapped to publish/subscribe semantics that realize the process control flow among activity agents. These agents are loosely coupled in the publish/subscribe layer, which makes our agent-based BPEL engine more flexible and customizable. Third, we present how to deploy processes into the agent network, initiate a process instance, and manage the process execution. The process deployment, execution and management are performed through the publish/subscribe layer taking advantage of the event-driven and the loosely coupled nature of the publish/subscribe infrastructure. Finally, we carry out a set of experiments comparing our distributed agent-based engine with a centralized orchestration scenario and a clustered scenario. The evaluation indicates that the benefit of the distributed approach is more
apparent under a higher process request workload, say over 300 requests per minute. In addition, the distributed approach is well suited to execute highly parallel processes that are not feasible in a centralized deployment.

Chapter 6 then simplifies the management of this process execution architecture. A cost model is developed to allow goals, such as minimizing bandwidth resources and process response times, to be independently specified on the executing process, and algorithms are devised to redeploy the executing process to satisfy the specified goals. A number of techniques are developed to overcome issues that stem from the lack of global knowledge. These include dampening the frequency of redeployment decisions, evaluating multiple points in the solution space simultaneously, and considering the sensitivity of a particular deployment to disturbances in the environment. Next, the dynamic execution engine is used not only to optimize the execution of business processes but also to optimize the monitoring of these processes. Based on the insight that monitoring a process can be modelled as a process itself, a systematic method to map SLAs to a set of monitoring agents is presented. This technique brings the same benefits of the distributed execution architecture to process monitoring, namely more efficient use of resources based on strategic deployment of monitoring components. Evaluations support the ability of the system to repeatedly adapt to changing runtime conditions to achieve a declaratively specified goal. In one workload, the system was able to save about 90% of the message traffic by adapting a process compared to an initially optimal, but static deployment.

Distributed protocols are developed to find and automatically compose services in the system. Chapter 7 proposes a new service discovery framework that leverages the properties of distributed content-based publish/subscribe systems. The framework supports three discovery models: static, dynamic and continuous. The static model is used to discover services with fixed attributes, the dynamic model for services with attributes that may be updated, and the continuous model allows for real-time notifications of newly
Chapter 10. Conclusions

registered services. All three models can co-exist in one system and complement one another. In addition, a similarity-based optimization algorithm is presented that utilizes publish/subscribe covering techniques to reuse the discovery results among different concurrent discovery requests. Thorough evaluations and analyses of the algorithms are presented under a variety of workloads. The experimental results show that the similarity optimization can substantially reduce the discovery costs in terms of the time to perform discoveries, the network traffic incurred by discovery in a distributed system, and the discovery processing overhead. Moreover, these benefits improve in scenarios where the discoveries exhibit increasing similarities.

Chapter 8 adds the ability to automatically compose a set of services into a process based on some user-defined criteria. It develops, to the best of our knowledge, the first distributed algorithm for automatic service composition. In a novel use of the publish/subscribe model, service interface specifications are mapped to content-based publish/subscribe messages in such a way that ordinary publish/subscribe matching reveals possible service compositions. Based on this mapping, a distributed process search algorithm is developed that exploits the distributed matching capabilities of content-based publish/subscribe systems. The processing of a search is shared by the publish/subscribe broker network, which determines possible service relationships, and a set of service agents that collaborate to prune this space and find matching processes or service compositions. The benefits of the distributed architecture include the avoidance of any single point of failure or performance bottleneck, scalability to large numbers of services, and the ability to parallelize searches across distributed resources. In detailed evaluations, an implementation of the distributed algorithm is compared to a centralized one in which all the brokers and service agents are deployed on one physical machine. The results indicate that the distributed algorithm typically scales better and is more stable with respect to the time needed to complete a search. This pattern holds under a variety of conditions, including large numbers of concurrent search requests, increasing results per
search, and growing number of registered services. The distributed approach, however, does impose a larger network traffic cost arising from communication overhead among the distributed components. Despite this overhead, the distributed scheme achieves superior performance by parallelizing searches among the resources in the system, including the brokers and service agents. This hypothesis is confirmed by experiments where there is little parallelism possible, such as when there are no concurrent search requests, in which cases the centralized algorithm achieves faster search processing latency.

Finally, Chapter 9 extends the WSLA specification to develop a more flexible model to author SLAs. The model is loosely coupled with the business process it is referencing, so that both the business process and SLA can be evolved independently. In addition, the proposed model is highly modular and extensible, allowing complex SLA contracts to be constructed by composing elementary ones. As well, a distributed architecture is developed, based on a publish/subscribe platform, to monitor an SLA. In the design, each SLA is composed of a number of automatically generated monitoring clients that coordinate with each other by exchanging events to verify if the SLA is being violated. The architecture is modular and supports the reuse of monitoring clients. Moreover, the SLAs are monitored by consuming events emitted by the process execution engine. As a consequence, there is no need to modify the process to support the monitoring of the associated SLAs, and SLAs can be modified without interrupting running processes.

## 10.2 Future work

The publish/subscribe extensions to support the movement of clients and manage unsub-criptions were developed in anticipation of the requirements of a distributed business process management platform and were utilized in this context. However, these core features may be useful in other scenarios as well and it would be interesting to evaluate their efficacy in other application domains.
In developing a distributed process execution engine, this thesis described how a BPEL process can be mapped to activity agents in the architecture. This exercise revealed that some of the processing required by these activities had to be carried by the application layer, that is, by the activity agents themselves. It would be interesting to extend publish/subscribe functionality so that more of this processing can be pushed to the publish/subscribe layer. Doing so simplifies application development and allows the publish/subscribe layer to optimize the processing. Furthermore, the architecture can be extended to support a model that is a hybrid of the distributed and clustered approaches. In this hybrid model, a process is decomposed into individual activities and some of these activities are replicated so that process instances can be load balanced among these replicas. New activity placement algorithms would also have to be devised to support this hybrid model.

The automatic service composition algorithm developed in this thesis could be extended with support for continuous search, in which the system will asynchronously report new service compositions that become available as new services are added to the system. This functionality corresponds nicely with the continuous service discovery model developed in this work. As well, both the service discovery algorithms could be more tightly integrated into the execution engine architecture so that processes can dynamically invoke services that are discovered over time. This means, for example, that a process invoking a currency exchange service can be dynamically bound to an instance of this service at run time as found by the service discovery component.
Bibliography


