Optimal and Robust Routing of Subscriptions for Unifying Access to the Past and the Future in Publish/Subscribe

by

Guoli Li

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

Copyright © 2010 by Guoli Li
Abstract

Optimal and Robust Routing of Subscriptions for Unifying Access to the Past and the Future in Publish/Subscribe

Guoli Li

Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2010

A flexible, scalable, and asynchronous middleware abstract is needed for business process management, which involves thousands of tasks and a large number of running instances of large business processes. The content-based publish/subscribe system is an ideal candidate to serve as enterprise service bus for these applications. In the publish/subscribe paradigm, information providers called publishers disseminate publications to all subscribers who have expressed interests by registering subscriptions through a loosely coupled interface. However, the traditional publish/subscribe paradigm only supports “stateless” subscriptions, that is, event correlation is ignored. Moreover, subscribers can only receive publications issued after their subscriptions. There are many application contexts, however, where access to publications from the past is necessary, such as for replaying a business process execution to debug it. Even more interesting uses arise when data from the past can be correlated with those in the future. Therefore, new languages and new functionalities are needed in the standard publish/subscribe model in order to support business process management.

A new subscription language PADRES SQL(PSQL) which can express event patterns and unify both historic and future views for subscribers. PADRES allows a subscriber to access data published both in the past and in the future. Furthermore, complex event detection happens in the broker network. The main difficulties of distributed event de-
tection are routing a composite subscription, including where and how to decompose the composite subscription, and routing the individual parts of the subscription. Our composite subscription routing decisions are based on a cost model which minimizes the routing and detection delay. An adaptive subscription routing protocol is proposed to determine efficient location with dynamic changing workloads. Padres also provides robust message delivery by exploring alternative paths in a cyclic overlay. Routing optimizations and efficient matching algorithms are studied to improve the performance of the extended publish/subscribe model.

With the above features, we propose the Niños system, the distributed business process execution architecture as a case study, which uses light-weight activity agents to carry out business process execution in a distributed environment. Niños proves that decentralized business process execution is the trend for next generation products, and the publish/subscribe model is ideal to serve as an enterpriser service bus (ESB) for distributed applications.
Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Hans-Arno Jacobsen, for leading his invaluable experience and guidance to this thesis, as well as all of my research at Middleware Systems Research Group, University of Toronto. I thank him for the many individual meetings he had with me, discussing everything from research to career choices, guiding my papers, and improving my talks.

Special thanks goes to Vinod Muthusamy, a co-conspirator in much of my research. We have spent years together writing papers and discussing research problems. Vinod’s insightful ideas about any given research problem is impressive to me during our discussions. His effort has shaped my research and working with him during my Ph.D has been a pleasure.

While at University of Toronto, I had the privilege of interacting with wonderful bright and talented Middleware Systems Research Group members: Alex Cheung, Chen Chen, Amer Farroukh, Patrick Lee, Bala Mainymaran, Reza Sherafat, Naweed Tajuddin, Young Yoon, and some alumnus Eli Fidler, Shuang Hou, Haifeng Liu, Zhengdao Xu, Alex Wun. Their advice, feedback on my research, and the friendship have made my Ph.D experience both more educational and more fun. It is a great pleasure to work with all of you.

I would like to thank my husband Zhan Wu, my dear son Raymond and my dear daughter Elaine for their love and support throughout my doctoral studies, and for being a constant source of amusement.

Finally, I would like to thank my wonderful mother, Yue’e Guo, and my sister, Guoying Li, for everything that they have given to me. They have stood by me in everything I have done, providing constant support encouragement, and love.
# Contents

1 Introduction  
1.1 Motivation 2  
1.2 Problem Statement 7  
1.3 Contributions 11  
1.4 Organization 15  

2 Background 17  
2.1 The Publish/Subscribe Model 17  
2.2 Content-based Routing and Optimizations 19  
2.3 The PADRES System 20  
2.4 Complex Event Processing 22  
2.5 Historic Data Access 23  
2.6 Business Process Management 25  

3 Related Work 31  
3.1 Publish/Subscribe Systems 31  
3.1.1 SIENA 32  
3.1.2 REBECA 33  
3.1.3 Hermes 33  
3.1.4 Gryphon 34  
3.1.5 Le Subscribe 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.6</td>
<td>PADRES</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Complex Event Processing</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1</td>
<td>SNOOP</td>
<td>37</td>
</tr>
<tr>
<td>3.2.2</td>
<td>GEM</td>
<td>38</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Event Processing Service</td>
<td>38</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Cambridge Event Architecture</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Stream Processing</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Distributed Workflows</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>The Subscription Language Model</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Subscription Language Features</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>The Subscription Language</td>
<td>54</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Basic Operations</td>
<td>55</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Requirements</td>
<td>56</td>
</tr>
<tr>
<td>4.2.3</td>
<td>PSQL Language</td>
<td>57</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Discussion</td>
<td>61</td>
</tr>
<tr>
<td>5</td>
<td>Robust Content-based Routing</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>Composite Subscription Routing</td>
<td>63</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Simple Routing</td>
<td>64</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Topology-based Routing</td>
<td>65</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Unsubscription of Composite Subscriptions</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Composite Event Detection</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>Adaptive Routing in General Overlays</td>
<td>77</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Challenges</td>
<td>77</td>
</tr>
<tr>
<td>5.3.2</td>
<td>TID-based routing</td>
<td>79</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Dynamic Publication Routing</td>
<td>82</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Evaluation</td>
<td>86</td>
</tr>
</tbody>
</table>
6 Historical Data Access

6.1 Architecture and Routing ........................................ 95
6.2 Database Partitioning ........................................ 97
  6.2.1 Publication Space Partitions .............................. 97
  6.2.2 Database Provisioning ................................... 101
  6.2.3 Dynamic Partitions ....................................... 107
  6.2.4 Partition Selection Cost Model .......................... 108
6.3 Evaluation ..................................................... 111

7 Cost Model .......................................................... 120

7.1 Cost Model ....................................................... 121
  7.1.1 Model Description ......................................... 121
  7.1.2 Example .................................................. 123
  7.1.3 Join Point Placement Cost Model ......................... 125
7.2 Evaluation for General Overlays ............................... 127
7.3 Evaluation for Historic Data Access ............................ 129

8 Case Study: Supporting BPEL ........................................ 132

8.1 Distributed Process Execution ................................. 133
  8.1.1 NIÑOS System Architecture ............................... 133
  8.1.2 Process Transformation .................................. 136
  8.1.3 Process Deployment ...................................... 145
  8.1.4 Process Execution ....................................... 148
  8.1.5 Process Management .................................... 149
  8.1.6 Architectural Benefits .................................. 151
8.2 Evaluation ..................................................... 154
  8.2.1 Experimental Setup ...................................... 154
  8.2.2 Request Rate ........................................... 154
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2.3</td>
<td>Web Service Delay</td>
<td>156</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Message Size</td>
<td>159</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Parallelism</td>
<td>162</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions</td>
<td>165</td>
</tr>
<tr>
<td></td>
<td>9.1 Summary and Discussion</td>
<td>165</td>
</tr>
<tr>
<td></td>
<td>9.2 Future Work</td>
<td>169</td>
</tr>
<tr>
<td>Appendices</td>
<td></td>
<td>170</td>
</tr>
<tr>
<td>A</td>
<td>BPEL Example: Loan Approval</td>
<td>171</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>177</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of BPEL Activities</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>Effect of Subscriber Distance on Delay</td>
<td>92</td>
</tr>
<tr>
<td>6.1</td>
<td>Publication Storage Time</td>
<td>111</td>
</tr>
<tr>
<td>7.1</td>
<td>Selection Factor</td>
<td>122</td>
</tr>
<tr>
<td>7.2</td>
<td>Subscription Cardinality</td>
<td>122</td>
</tr>
<tr>
<td>A.1</td>
<td>Agent messages for loan application process in Figure A.1</td>
<td>175</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 PADRES Broker Network ................................. 20
2.2 PADRES Router Architecture ............................. 22
2.3 Enriched Rule-based Matching Engine in PADRES ........ 23

4.1 A Composite Subscription ............................... 55

5.1 Algorithm of Building a Destination Tree ................. 66
5.2 Algorithm of Decomposing a Composite Subscription ....... 67
5.3 A Simple Composite Subscription Routing Example ......... 68
5.4 A Complicated Composite Subscription Routing Example ..... 69
5.5 Algorithm of Unsubscribing Subscriptions at the First Broker ... 70
5.6 Unsubscription at the First Broker ....................... 71
5.7 Algorithm of Unsubscribing Distributedly Decomposed Subscriptions .... 72
5.8 Unsubscription of Distributedly Routed Composite Subscriptions .... 73
5.9 Composite Subscription Matching .......................... 74
5.10 Adding a Fact $e_{32}$ to Rete Network .................. 76
5.11 Cyclic Routing of Subscriptions .......................... 77
5.12 TID-based Subscription Routing .......................... 78
5.13 Multiple Publication Routing Paths ....................... 80
5.14 End to End Notification Delay in Dense vs. Sparser Topologies ..... 87
5.15 End to End Notification Delay with More Publishers ........ 88
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.16</td>
<td>End to End Notification Delay on PlanetLab</td>
<td>89</td>
</tr>
<tr>
<td>5.17</td>
<td>End to End Notification Delay with Failures</td>
<td>94</td>
</tr>
<tr>
<td>6.1</td>
<td>Architecture of Historic Data Access</td>
<td>95</td>
</tr>
<tr>
<td>6.2</td>
<td>Partition Example</td>
<td>100</td>
</tr>
<tr>
<td>6.3</td>
<td>Overlaps</td>
<td>108</td>
</tr>
<tr>
<td>6.4</td>
<td>Query Processing Time</td>
<td>112</td>
</tr>
<tr>
<td>6.5</td>
<td>Partition Sinking Traffic</td>
<td>113</td>
</tr>
<tr>
<td>6.6</td>
<td>Notification Delay</td>
<td>113</td>
</tr>
<tr>
<td>6.7</td>
<td>Delay Range</td>
<td>116</td>
</tr>
<tr>
<td>6.8</td>
<td>Partitioning</td>
<td>116</td>
</tr>
<tr>
<td>6.9</td>
<td>Hybrid Atomic Subscription</td>
<td>118</td>
</tr>
<tr>
<td>6.10</td>
<td>Hybrid Composite Subscription</td>
<td>118</td>
</tr>
<tr>
<td>7.1</td>
<td>Dynamic CS routing</td>
<td>123</td>
</tr>
<tr>
<td>7.2</td>
<td>CS traffic</td>
<td>127</td>
</tr>
<tr>
<td>7.3</td>
<td>CS delay</td>
<td>127</td>
</tr>
<tr>
<td>7.4</td>
<td>Composite Subscirption</td>
<td>130</td>
</tr>
<tr>
<td>7.5</td>
<td>Dynamic Workload</td>
<td>131</td>
</tr>
<tr>
<td>8.1</td>
<td>NIÑOS Distributed Business Process Execution Architecture</td>
<td>133</td>
</tr>
<tr>
<td>8.2</td>
<td>Conceptual System Architecture</td>
<td>135</td>
</tr>
<tr>
<td>8.3</td>
<td>BPEL While Activity</td>
<td>138</td>
</tr>
<tr>
<td>8.4</td>
<td>BPEL Pick Activity</td>
<td>138</td>
</tr>
<tr>
<td>8.5</td>
<td>BPEL Compensate Activity</td>
<td>140</td>
</tr>
<tr>
<td>8.6</td>
<td>BPEL Switch Activity</td>
<td>140</td>
</tr>
<tr>
<td>8.7</td>
<td>BPEL Flow Activity</td>
<td>142</td>
</tr>
<tr>
<td>8.8</td>
<td>Process Deployment and Execution Monitor</td>
<td>145</td>
</tr>
<tr>
<td>8.9</td>
<td>Inter-organizational Deployment</td>
<td>147</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Enterprise applications are increasingly being architected in a service-oriented architecture (SOA) style, in which modular components are composed to implement the business logic. The properties of enterprise applications, such as the loose coupling among the modules, are 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 profusion 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.

Distributed enterprise applications involve processing of data distributed among large numbers of data sources and sinks. Typically, the distributed data sources are updated independently and the data sinks are interested in changes as they occur. Consequently, there is a growing need for a new kind of middleware system that facilitates such applications. The middleware system enables fine-grained filtering, loose coupling among multiple data sources in a scalable and efficient manner. A publish/subscribe system consists of a set of publishers and subscribers who exchange information asynchronously in a distributed environment. Publishers send events to the system as publications and subscribers express their interests in these events by issuing subscriptions. The publish/
subscribe system matches or routes relevant publications to interested subscribers. Instead of address-based routing, messages in content-based publish/subscribe systems are routed based on their content. Publishers and subscribers are loosely coupled in space and time, as they have no knowledge of each other. A content-based publish/subscribe system provides a flexible and extensible environment for information exchange. A large variety of emerging applications benefit from content-based publish/subscribe systems. These applications include RSS feed filtering [79], stock-market monitoring engines [83], system and network management and monitoring [66, 25], algorithmic trading with complex event processing [45, 77], business process management and execution [81, 8], business activity monitoring [25], workflow management [22], and service discovery [40].

In the following sections, we describe the characteristics of these distributed emerging applications which require functionalities that are missing in the standard publish/subscribe paradigm, as they pertain to the motivation of this thesis.

1.1 Motivation

There has been great interest in the publish/subscribe paradigm over the past few years. In academia, the focus has been on developing efficient matching algorithms for centralized publish/subscribe [24] and on content-based routing [14, 65, 22] for distributed publish/subscribe architectures. In industry, several standards have emerged that define common application programming interfaces for publish/subscribe-style interactions, such as the CORBA Notification Service [69], the OMG Data Dissemination Service Specification [73], and the Java Messaging Service [36].

With the growth of the Internet and Web services, enterprise applications have become distributed and globalized. The unique features of these applications require emerging technologies, and bring major challenges to current content-based publish/subscribe systems. We discuss major enterprise application characteristics below.
Complex Event Processing

Event processing is the ability to detect and respond to events (or activities) occurring across the enterprise. Event processing, in general, adds a dynamic dimension to an application by enabling insight that facilitates decision-making based on observations about things that happen in the enterprise. It can help in identifying trends, threats, opportunities, and business situations that need action. Event processing is not a new concept. For the last forty years, organizations have been utilizing a form of simple event processing to detect and respond to a single source, or homogeneous, event type.

In recent years, the complex event processing (CEP) attracts more attention in both academia and industry, for it tackles problems of greater complexity that traditional event processing could not handle. For instance, with CEP you can “analyze, correlate and summarize low-level events into higher-level events suitable for notifying people in human terms or for triggering automated processes.” CEP employs techniques such as detecting complex patterns across many events, using rule processing algorithms for event correlation and abstraction, using event hierarchies and relationships among events. Analysis of causality, membership, timing, and event-driven processes is core capability of these technologies. Business Event Processing (BEP), is the next generation of event processing, extending the capabilities and tools of technologies like CEP and event stream processing (ESP) to business users to define and detect situations in business context for rapid response to opportunities and threats.

In order to make the CEP possible in content-based publish/subscribe systems, the major challenge is the lack of a powerful subscription language for subscribers to describe their complex interests and the inability to handle large amounts of dynamically changing information and interest specifications (i.e., publications and subscriptions). So in this work, first, an appropriate subscription language needs to be designed which offers the right view over available events to enable event coordination. Second, event correlation requires the detection of disparate events. In publish/subscribe this is based on rout-
ing subscriptions throughout the broker network and, finally, on routing computations realized on a single publish/subscribe broker.

**Data Management**

Distributed business applications involve a large amount of data and events naturally distributed in space and in time. Applications need to manage the information in order to, for example, create reports and analyze the data to make quality business decisions: storing data effectively so that it can be accessed and used efficiently; analyzing data to show trends, relationships, and patterns that can help a business grow or create profit; moving data from one place to another inexpensively and safely; maintaining data integrity among replicas, etc.

Distributed database system (DDBS) [72] is particularly designed for this purpose. DDBS technology is the union of *database system* and *computer network* technologies. Database systems have taken us from a paradigm of data processing in which each application defined and maintained its own data to one in which the data is defined and administered centrally. This new orientation results in data independence, whereby the application programs are immune to changes in the logical or physical organization of the data, and vice versa. While “a distributed database as a collection of multiple, logically interrelated databases distributed over a computer network.”, a distributed database management system is then defined as “the software system that permits the management of the DDBS and makes the distribution transparent to the users” [72]. The advantages of distributed database systems range from sociological reasons for decentralization [23] to better economics, including transparent management of distributed and replicated data, reliability through distributed transactions, improved performance, and easier system expansion. However, all these promises of DDBS require that data must already exist and follow predefined schema.

While for business applications, detecting events and processing data in real-time are
critical to make business decisions. In recently years, *continuous query* paradigm [9] has attracted much attention, in which users register logical specifications of interest over streaming data sources, and a continuous query engine filters and synthesizes the data sources to deliver streaming, unbounded results to users [21, 58], instead of sinking all data into database and processing them afterwards.

Distributed database systems focus on existing data, while continuous queries process streams coming in the future. However, in business applications, it is even more interesting when data from the past can be correlated with those in the future\(^1\). For example, a credit-card company may need to know if a client’s usage pattern differs significantly from those on the same day last week. While such management can be performed periodically with traditional databases or combine the database techniques with the stream processing techniques, the publish/subscribe model can detect these complex patterns in real-time and immediately notify subscribers as they occur. The challenge is to support access to both historic and future data through a unified publish/subscribe interface while still preserving the desirable properties of the publish/subscribe model.

**Robust and Efficient Message Delivery**

Distributed business applications operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users might involved. Due to the growth of Web services, business applications usually deploy over the Web and run across the Internet. The number of users might involved in such applications has increased dramatically. Moreover, computer systems sometimes fail. When faults occur in hardware or software, systems may produce incorrect results or they may stop before they have completed the intended computation

\(^{1}\)Data available before a query is considered historic. The past and future are with respect to the time the query is issued.
or services. For example, a web service request message might be lost due to network failures. While every message in a business application is critical, robust message delivery is an important requirement in this kind of applications. As a result, serving as a messaging substrate, the underlying publish/subscribe system should be able to tolerate failures (e.g., using redundant components to tolerate failures). The design of scalable distributed middleware has the following challenges.

- Controlling the cost of physical resources: As the demand for a resource grows, it should be possible to extend the system in scale and in functionalities, at reasonable cost, to meet it. For example, a popular publication message might be interested to many subscribers. It must be possible to eliminate redundant message routing and matching to avoid the performance bottleneck. In general, for a publish/subscribe system with $n$ subscribers having the same subscription, instead of routing all $n$ subscriptions to potential publishers, duplicate subscriptions can be removed from common paths to save network traffic and matching time per hop.

- Tolerating failures: Distributed systems provide a high degree of availability in the face of hardware and software faults. The availability of a publish/subscribe system is a measure of the proportion of time that it is available for guaranteed message delivery. For example, in a general overlay, there is always at least two different routes between any two nodes, if failures happen on one of the routes, a message can still be delivered to its destination through other alternative routes.

- Adaptive to network conditions: publishers and subscribers are exchanging messages spontaneously. There is a possibility that a broker on a common path of many clients will be overloaded due to the overall traffic. The publish/subscribe system throughput might be effected because of the overloaded broker. It must be possible to balance the workload among brokers and route messages around congestions and failures.
1.2 Problem Statement

Composite Subscriptions

Many current publish/subscribe systems only support primitive subscriptions, that is, subscriptions on atomic events, and lack the ability to express complicated event patterns. However, in real-world applications, subscribers may be interested in complicated event patterns. For example, in a network management system, several primitive events raised by network devices may indicate an actual fault in the network. The network manager wants to be notified only if all the primitive events composing the failure occur. Diagnosing the specific cause must be carried out as quickly as possible so that action is taken and faults can be repaired. Large-scale publish/subscribe systems need to support composite event detection in order to quickly and efficiently notify their clients of new relevant information in the network. With composite subscriptions, the manager may subscribe to an event pattern which indicates a network failure, and be notified only when the pattern is detected.

The objective of composite subscription is to provide a higher level view for subscribers by enriching the expressiveness of the subscription language. Composite subscription routing policies and composite event detection are designed to notify subscribers when their composite subscriptions are satisfied. composite subscription consists of several atomic subscriptions linked by logical or temporal operators. An atomic subscription refers to the traditional notion of a subscription in publish/subscribe and is matched by a single publication event; a composite subscription is matched by a set of independent events potentially occurring at different locations and times. Composite subscriptions have three advantages. First, subscribers can process events from a higher level of view. This makes publish/subscribe systems suitable for more application domains. Second, subscribers receive fewer messages. Without composite subscriptions, the subscriber has to subscribe to all events generated by the publishers in order to receive the proper
information, and special logic is needed on the subscriber side to detect the occurrence of a composite event. The subscriber would be overwhelmed by an excessive number of primitive events, most of which are irrelevant and could be filtered out before reaching the subscriber. Third, composite event detection is efficient. Detecting the occurrence of composite event patterns on the subscriber side is unnecessarily complex and error-prone. Most of the detection is redundant among subscribers who have similar interests. With composite subscriptions, the detection of composite events occurs within the network, and the detection results can be shared among clients. The network traffic and detection efficiency are improved. With the support of composite subscriptions, it is possible to handle large numbers of events generated in an Internet-scale system.

**Historic Data Access**

A limitation of the traditional publish/subscribe model is that it only delivers to subscribers those publications produced after the subscription was issued. It is a model to query the future. There are many application contexts, however, where access to publications from the past is necessary, such as for auditing purposes, replaying a business process execution to debug it, or tracing system events to perform root cause analysis. Even more interesting uses arise when data from the past can be correlated with those in the future. For example, an algorithmic trader may wish to be notified when a stock behaves as it did during a recent economic downturn (incoming stock quotes are being correlated with those from the past). While such analysis can be performed periodically with traditional databases, the publish/subscribe model can detect these complex patterns in real-time and immediately notify subscribers as they occur.

Another scenario involves a border security agency that prevents sensitive goods, such as radioactive material, from entering the country [67]. Containers entering at around 30 border points by ship, rail, and truck are scanned in sensing stations, where traces of radioactive material trigger alarms. There are, however, genuine goods that set off the
sensors, and to filter out false alarms, the serial numbers of radioactive containers are correlated with shipping manifests submitted by the shipping company ahead of time.

The challenge is to support access to both historic and future publications through a unified publish/subscribe interface while preserving the desirable properties of the publish/subscribe model. For example, in content-based publish/subscribe, addresses of participants are not available, so directly querying a database is not an option. The client should not even know which database to query. Also, composite subscriptions [52] that allow correlations, or joins, across publications should work with any combination of historic and future data. Furthermore, to preserve the scalability of distributed publish/subscribe architectures, the historic data repository should not be centralized. But replicating and distributing the repositories present further challenges such as replica synchronization.

Robust Content-based Routing

Most existing publish/subscribe systems [22, 28, 70] are based on an acyclic overlay broker network. With only one path between any pair of brokers or clients, content-based routing is greatly simplified. Despite this success, an acyclic overlay offers limited flexibility to accommodate changing network conditions, is not robust with respect to broker failures, and introduces complexities for supporting other protocols, such as failure recovery and load balancing. For example, since only one path exists between any pair of clients, an acyclic overlay cannot accommodate routing around congested, overloaded, or failed brokers. Furthermore, solutions for failure recovery and topology reconfiguration in an acyclic overlay can be complex since their repair actions must maintain the acyclic property of the overlay [74]. Maintaining the acyclic property is difficult since a broker often only knows about its direct neighbors and not the entire topology.

However, supporting general overlay topologies requires changes to the standard content-based routing protocol in order to avoid routing messages in cycles. Consider a
topology graph $G = (V, E)$ with vertices $V$ and edges $E$. Broadcasting a single message in $G$ will generate $\frac{|E| - |V|}{|E|} \%$ of redundant messages. For instance, in a 500 broker topology with an average connectivity of 10 neighbors, a single advertisement induces 2500 messages, 80% of which must be discarded.

In such general overlays, alternative routing paths are available, and diverse data sources exist. The challenge is to support great flexibility for selecting an optimal routing path based on some optimality criteria or utility function, something not possible in acyclic overlays, where at most one path exists between any data source and sink pair. By allowing for general overlay topologies, the content-based publish/subscribe protocol in this work also provides a foundation for potentially facilitating the support of other features such as failure recovery, load balancing, path reservation, or splitting message streams across multiple paths.

Workflow Management

Workflow management is a fundamental component of any data-centric operation, and any gains made in this area have far reaching impact and benefits for enterprises. Workflow management solutions have existed for a number of decades [6, 30]. The traditional approach to workflow management is based on using a centralized server to coordinate and schedule tasks [35]. This architecture is inherently limited. The centralized monitoring and control point constitutes a single point of failure in the system. It also forms a bottleneck for system monitoring and control, and can significantly affect overall application operation and scheduling performance. Moreover, resources distributed across multiple platforms cannot be federated with ease, and distributed tasks are difficult to handle. The geographical distribution of tasks calls for a distributed middleware architecture, and the publish/subscribe paradigm is an ideal candidate.

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. These 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. The processes are inherently distributed using department-level processes for manufacturing, warehouse and order management. Each of these processes utilizes 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, 16 sales centers, and many retailers. Thousands of instances of these processes are executing concurrently at any point in time. Such large processes involving dozens of collaborating parties is a natural fit for a distributed execution architecture.

A distributed architecture removes the scalability bottleneck of a centralized orchestration engine, and is congruent with the inherently distributed enterprise business processes where geographically dispersed parties communicate across administrative domains. The challenges are supporting flexible mappings of the orchestration process onto heterogeneous platforms and resources, permitting the system to shape itself from a centralized to a fully distributed configuration, furthermore, offering additional efficiencies by allowing portions of processes to be executed close to the data they operate on, thereby conserving data and control traffic.

1.3 Contributions

The work of this thesis has been integrated into PADRES which has been developed in Java by Middleware Systems Research Group (MSRG), University of Toronto. PADRES
is funded in part by CA, CFI, IBM, NSERC, OCE, OIT, and Sun. The contributions of this thesis are the following:

- The subscription language in PADRES is extended to support composite subscriptions [52], with which clients can express constraints, correlations, projections and aggregations on any combination of future and historic publications in a unified manner [55]. An expressive SQL-like subscription language called PADRES SQL (PSQL) is proposed to unify access to publications from the past and the future. Evaluating future PSQL subscriptions is straightforward, and subscriptions described in PSQL are easier to map to SQL and evaluate in a relational database. An SQL-like language also afforded a familiar construct to express simple joins (composite subscriptions), and notification semantics such as projections and aggregation constraints, which are not typically supported by publish/subscribe systems.

- We developed significant extensions to the standard content-based routing protocol to enable message routing in general overlay topologies [54]. The design preserves the original simple publish and subscribe interface to clients, and does not require changes to a broker’s internal message matching algorithms, allowing the approach to be easily integrated into existing systems. The solution applies to advertisement-based and to subscription-based routing, and exploits redundant routing paths so that publications can be routed to subscribers more optimally, for example, based on assessing load conditions on links. An interesting byproduct of the approach is that message routing can be significantly improved by performing matching only once per message, as opposed to existing approaches that match messages at each broker in the overlay.

- An advanced publish/subscribe model is proposed to support the unified retrieval of data published before and after a subscription is issued [48, 55]. An architecture
to evaluate historic subscriptions, future subscriptions, and the hybrid of both is presented. The architecture employs a set of databases to store publications and responds to historic subscriptions. Concepts from distributed databases are used but the publish/subscribe model differs sufficiently from the relational model to preclude a direct transfer of the ideas. The database architecture supports a range of distributed and replication strategies with the ability to arbitrarily assign portions of the data space to one or more databases. An algorithm is presented to uniformly partition the data space, and a synchronization protocol is developed to ensure replica consistency among replicas.

- We propose a general cost model for adaptive subscription routing. Compare to different composite subscription routing policies such as simple routing and topology-based routing, the goal of the adaptive policy is to place the composite event detector close to data sources so that event patterns are detected with minimized cost [54]. With a cost model, the policy can dynamically determine the optimal points for composite subscription correlation operations based on traffic volumes and network conditions, when exposed to multiple routing paths and diverse data sources.

- NIÑOS [53, 57, 40, 41] is a distributed orchestration architecture for business processes. The contributions include, the design of the NIÑOS distributed business process execution architecture based on the flexible PADRES publish/subscribe layer; a procedure to map standard Business Process Execution Language (BPEL) processes, including the complete set of BPEL activities, to a set of distributed NIÑOS agents, with control flow realized using decoupled publish/subscribe semantics; and an evaluation of the NIÑOS orchestration engine that demonstrates its improved scalability over a centralized engine.

NIÑOS utilizes and exploits the rich SOA enterprise service bus (ESB) capabilities
Chapter 1. Introduction

of the PADRES [27] distributed content-based publish/subscribe routing infrastructure. All communication in the system occurs as publish/subscribe interactions, including process coordination among the agents, control and monitoring. This decouples the agents, which now only need to be aware of one another’s content-based addresses, which simplifies agent reconfiguration and movement, and seamlessly allows 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 agents, 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. Agents 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.

- We also applied the content-based routing protocol and the covering and merging optimizations to XML-based data dissemination networks [49, 51, 50]. First, we adapt the use of advertisements to optimize data dissemination. While this idea is common in the publish/subscribe literature, it is not clear how to extend the concepts to the data model of XML. We demonstrate how to use the XML Document Type Definition (DTD) to generate advertisements about the information a data producer is going to publish. We distinguish between a non-recursive and a recursive case depending on the DTD defining the data emitting source. We then develop advertisement-based routing algorithms for both cases. Second, we propose a novel data structure to maintain XPEs by identifying the covering rela-
tions among them. We present covering algorithms for XPEs to reduce the routing table size stored at each router and speed up routing computation in the routers. Third, we present an optimization of merging similar XPEs to further reduce routing computation. Characteristics of the merging technique, which are not fully discussed in previous works, are explored in this thesis, especially for imperfect merging. Imperfect merging allows the creation of a more concise routing table than perfect merging by introducing some false positives. A parameter, degree of imperfectness, is defined to evaluate an imperfect merger and balance the trade-off between routing table size and false positives. Moreover, an advertisement-based optimization for imperfect merging is discussed. These techniques are applied to both predicate-based languages [49] and XML-based languages [51, 50].

- We further explored content-based routing algorithms using Binary Decision Diagram (BDD) [49]. A novel data structure and algorithms based on BDD unify publication routing, subscription covering and subscription merging. We also extend the work to support subscriptions with arbitrary boolean expressions for atomic and composite subscriptions [56].

1.4 Organization

The rest of this thesis is organized as follows. In Chapter 2, we review the publish/subscribe communication model as it pertains to this thesis, and define the key terminology used in the thesis. Chapter 3 discusses projects that are related to the techniques discussed in this thesis. We put the thesis in the context of complex event processing, continuous query, and publish/subscribe systems.

The subscription language model is presented in Chapter 4. We introduce the features of the subscription language and composite subscription routing policies. The strategies for event consumption in composite event detection are presented as well.
Chapter 5 discusses several optimizations of the content-based routing protocol. These optimizations and extensions provide more robust and efficient message delivery service for publish/subscribe clients. We develop significant extensions to standard content-based routing protocols to enable message routing in general overlay topologies, which exploits redundant routing paths so that publications can be routed to subscribers more optimally based on assessing load conditions on links.

Chapter 6 introduces to the publish/subscribe model the capability to uniformly access data produced in the past and the future. This new model can filter, aggregate, correlate and project any combination of historic and future data. A flexible architecture is proposed consisting of distributed and replicated data repositories that can be provisioned in ways to tradeoff availability, storage overhead, query overhead, query delay, load distribution, parallelism, redundancy and locality. Evaluations in a distributed testbed show that different provisioning policies perform better with read or write heavy workloads.

Chapter 7 proposes a cost model to minimize the routing and detection delay for subscriptions. An adaptive subscription routing protocol based on the cost model is proposed to determine efficient locations for composite event detection with dynamic changing workloads.

Chapter 8 presents a distributed agent-based orchestration engine NIÑOS in which several light-weight agents execute a portion of the original business process and collaborate in order to execute the complete process. The complete set of standard BPEL activities are supported in NIÑOS. Evaluations demonstrate that agent-based execution based on publish/subscribe scales better a non-distributed approach. Chapter 9 summarizes the contributions of this thesis and outlines areas of future work.
Chapter 2

Background

In this chapter, we describe the publish/subscribe interaction model. We also define the appropriate terminology and highlight the key characteristics of the PADRES system developed by Middleware Systems Research Group at University of Toronto.

2.1 The Publish/Subscribe Model

The publish/subscribe model is a good fit for selective information dissemination applications [3, 13, 22, 28, 52, 71, 79]. A publish/subscribe system is composed of loosely coupled clients that communicate through an intermediary. Data model and query language are used as a declarative and data centric way to specify connections between clients. The key piece of a publish/subscribe system is a broker, which executes the queries (e.g., subscriptions), and distributes data (e.g., publications) between clients. Loosely coupled systems are flexible to changes because their constituent clients communicate indirectly. This loosely coupling and decoupling features can enable the design for large and distributed systems that interoperate through simple publish/subscribe invocations. A large variety of emerging applications benefit from the expressiveness, the filtering, the distributed event correlation, and the complex event processing capabilities of the publish/subscribe model. These applications include information dissemination [71, 17], RSS feed filter-
ing [79], stock-market monitoring engines [83], system and network management and monitoring [66, 25], algorithmic trading with complex event processing [45, 77], business process management and execution [81, 8], business activity monitoring [25], workflow management [22], and service discovery [40].

In publish/subscribe paradigm, information producers (e.g., publishers) submit data as publications to the system and information consumers (e.g., subscribers) indicate their interests by submitting subscriptions. A subscription has a notification set, which indicates a set of potential publications that would match the subscription. On receiving a publication, the publish/subscribe system determines the subset of matching subscriptions and notifies the appropriate subscribers.

While publish/subscribe was first implemented in centralized client-server systems, current research focuses mainly on distributed versions. The key benefit of distributed publish/subscribe is the natural decoupling of publishers and subscribers. Since the publishers are unconcerned with the potential consumers of their data, and the subscribers are unconcerned with the locations of the potential producers of interesting data, the client interface of the publish/subscribe system is simple and intuitive. There are several different classes of publish/subscribe systems. In channel-based publish/subscribe [34], publishers publish data to channels and subscribers subscribe to particular channels and receive all the publications in these channels. The topic-based publish/subscribe [36, 17, 78] has a topic associated with each publication which indicates the region of interest for contained. Clients subscribing to a particular topic would receive all publications with the indicated topic. Topics are similar to the notion of groups used in the context of group communication [11]. More powerful topic-based systems allow for a hierarchy of topics [62, 80]. In that case, subscribers who subscribe to a particular topic receive all publications of that topic and its descendants. Content-based publish/subscribe systems add significant functionality by allowing subscribers to specify constraints on actual data within a publication. In contrast to the channel-based and topic-
based approaches, publications are classified according to their content. SIENA [13], REBECA [64], Gryphon [70], and PADRES [27] are some well-known content-based publish/subscribe prototypes.

2.2 Content-based Routing and Optimizations

Content-based publish/subscribe systems typically utilize content-based routing in lieu of the standard address-based routing. Messages in content-based routing are routed from source to destination based entirely on the content of the messages. Since publishers and subscribers are decoupled, a publication is routed towards the interested clients without knowing specifically where those clients are and how many such clients exist. Effectively, the content-based address of a subscriber is the set of subscriptions issued.

In a content-based publish/subscribe system with a predicate-based language, for example, a publication \( P \) is a set of attribute-value pairs. Formally, \( P \) has a set of attributes \( \{a_1, a_2, ..., a_i, ..., a_n\} \), and is described as \( P = \{(a_1, v_1), (a_2, v_2), ..., (a_n, v_n)\} \). A subscription \( S \) is specified as relations between attributes and values. It is a conjunction of a set of predicates which are also called attribute filters. A predicate [attribute, operator, value] is a constraint on the value of the attribute. Different subscription languages are studied in publish/subscribe systems as well. Bittner [12] proposed a language model where a subscription is an arbitrary Boolean function of predicates; Hou and Li [38, 51] supported XPath queries and XML documents in the publish/subscribe paradigm. In this thesis, we propose a more expressive subscription language, called PSQL [55], supporting composite subscriptions and unifying future and historic publication queries.

In addition to publications and subscriptions, content-based routing networks can use advertisements [64, 13], which are indications of the data that publishers will publish in the future. Advertisements are used to form routing trees along which subscriptions are
propagated. An advertisement has the same format as a subscription. Advertisements avoid broadcasting subscriptions in the network, since subscriptions are only routed to the publishers who advertise what the subscribers are interested in. Publications trace back along the path set up by subscriptions to interested subscribers. $P(S)$ refers all publications that match $S$; and $Attr(S)$ is used to describe the attribute set of subscription $S$.

### 2.3 The Padres System

![Diagram of Padres Broker Network](image)

The Padres system consists of a set of brokers connected by a overlay network, as shown in Figure 2.1. The overlay network forms the basis for message routing. Each Padres broker acts as a content-based router to route and match publish/subscribe messages. A broker only knows its direct neighbors, and the overlay information is stored in the Overlay Routing Tables (ORT) at the broker. Clients connect to brokers using various binding interfaces such as Java Remote Method Invocation (RMI) and Java Messaging Service (JMS). Publishers and subscribers are all publish/subscribe clients to the overlay.
A publisher issues an advertisement before it publishes. Advertisements are effectively flooded to all brokers along the overlay network. A subscriber may subscribe at any time. The subscriptions are processed according to the Subscription Routing Table (SRT), which is built based on the advertisements. The SRT is essentially a list of [advertisement, last hop] tuples. If a subscription intersects an advertisement in the SRT, it will be forwarded to the last hop broker the advertisement came from. Subscriptions are routed hop by hop to the publisher, who advertises information of interest to the subscriber. Meanwhile, the subscription will be used to construct the Publication Routing Table (PRT). Like the SRT, the PRT is logically a list of [subscription, last hop] tuples, which is used to route publications. If a publication matches a subscription in the PRT, it will be forwarded to the last hop broker of that subscription until it reaches the subscriber. A diagram showing the overlay network, SRT and PRT is provided in Figure 2.1. In this figure, step 1) an advertisement is propagated from $B_1$. Step 2) a matching subscription enters from $B_2$. Since the subscription overlaps the advertisement at broker $B_3$, it is sent to $B_1$. Step 3) a publication is routed along the path established by the subscription to $B_2$.

Each broker consists of an input queue, a router, and a set of output queues, as shown in Figure 2.2. A message first goes into the input queue. The router takes the message from the input queue, matches it against existing messages according to the message type, and puts it in proper output queues which indicate different destinations. Other components are provided for advanced features. For example, the controller component provides an interface for a system administrator to manipulate a broker (e.g., shut down a broker, inject a message into a broker etc.); the monitor component maintains the statistic information (e.g., monitor the incoming message rate, calculate average queueing time and matching time etc.). Once the broker is overloaded (e.g., the incoming message rate is above a certain threshold), a load balancer will trigger our offload algorithms to balance the traffic among brokers. A failure detector collects the topology and the subscription routing information continuously. At the same time, it is monitoring the communicating
peers. Once a failure is detected, a recovery procedure is triggered in order to guarantee message delivery in present of failures.

![Figure 2.2: PADRES Router Architecture](image_url)

### 2.4 Complex Event Processing

In traditional publish/subscribe systems, when a publication reaches a broker, the broker matches the publication against its databases of subscriptions and forwards the publication along the appropriate network links. Each publication match is an independent stateless operation. Composite subscriptions, which consist of primitive (atomic) subscriptions, are analogous to composite event detectors in event-processing systems. A composite subscription is matched only after all component atomic subscriptions are satisfied. That is, a composite event pattern in the publish/subscribe system occurs and is detected. Composite events are a key concept in the event-processing context and have not received much attention in the publish/subscribe literature. Some distributed publish/subscribe architectures such as Hermes [75] and Gryphon [70] provided only parameterized primitive events, and lacked the composite event detection module in the internal publish/subscribe system, which is responsible for disseminating events.
SIENA [13] supports restricted event patterns, but it does not define a complete pattern language. There has been some work in building a distributed composite event detection framework on top of publish/subscribe systems [77], but the detection is not in the publish/subscribe layer.

In this thesis, we extend the subscription language to support composite subscription in the rule-based matching engine. By mapping a composite subscription to a rule expression in PRT, the matching engine acts as a composite event detector. The composite subscription routing policy module determines the location of composite event detections. Figure 2.3 shows the enriched rule-based matching engine.

**2.5 Historic Data Access**

PADRES unlike existing content-based publish/subscribe systems, allows the subscriber to subscribe to data published in both the future and the past. For future publications, PADRES uses the standard publish/subscribe messaging paradigm. Historic databases are attached to the brokers through a database binding. Databases store publications as they are published. Later, upon receiving a request for the historic data, the brokers re-publish relevant publications from their databases. Since, in content-based publish/
subscribe no direct addresses of participants are available, directly querying the databases for past data is impossible. Furthermore, it is not even possible for the client to know which database to query. Historical queries are routed to associated databases according to their content, and get the historic publications for clients. From the client’s point of view, PADRES transparently delivers both past and future publications in the same manner (typically a Java callback). The powerful subscriptions supported in PADRES also allow the client to correlate past and future publications through temporal joins.

From the subscriber’s perspective, the historic data access is performed by adding time predicates to standard subscriptions. If the specified time range includes some amount of time previous to the query time (“now”), the historic portion of the query will be split off and sent along the advertisement tree toward the appropriate data source(s). Since the historic databases advertise the content stored in them, the historic subscription will automatically be sent to the correct database(s).

The databases, upon receiving the historic subscription, form an appropriate SQL query, retrieve the results from the data store, and re-publish the matching publications, which are routed to the subscriber along the subscription tree. Delivery of the historic publications happens in the same manner as standard (future) publications.

Simple historic queries are primitive subscriptions formatted as in a standard content-based publish/subscribe system. For example, \([\text{class}, \text{eq}, \text{trigger}] [\text{appl}, \text{eq}, \text{payroll}] [\text{gid}, \text{isPresent}, \text{anyID}]^1 [\text{time}, <, \text{now}+\text{1hr}] [\text{time}, >, \text{now} - \text{1hr}]\) matches all the trigger publications for application \text{payroll} published in a two hour window beginning one hour ago.

Complex historic queries involve the use of composite subscriptions. These queries can be significantly more powerful. For instance, \([\text{class}, \text{eq}, \text{job\_status}] [\text{appl}, \text{eq}, \$y] [\text{gid}, =, \$x] \text{AND} [\text{class}, \text{eq}, \text{trigger}], [\text{appl}, \text{eq}, \$y] [\text{gid}, =, \$x]\)

\(^1\text{isPresent}\) is an abbreviation of \text{isPresent}, an operator supported by PADRES. It means the attribute could be any value of its data type.
[time, <, now] matches the past triggers and job_status publications, which have the same generation id and application name. The shared variables $x$ and $y$ cause the join. Perhaps the subscriber wants to monitor executions of all applications triggered in the past.

In this thesis, we propose an advanced publish/subscribe architecture to support the historic data access functionality, including a new SQL-like subscription language, a publication space partitioning algorithm, and a set of partition replica management policies.

### 2.6 Business Process Management

Enterprise applications are increasingly being architected in a service-oriented architecture (SOA) style, in which modular components are composed to implement the business logic. The properties of such applications, such as the 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 profusion 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.

An important part of these enterprise applications is a language that allows one to author such applications by composing available services on the Internet. BPEL [8] is perhaps the most accepted such language in the Web services community. BPEL processes are specified using activities that implement constructs such as conditionals, loops, sequential and parallel execution, and exception handling. In addition, BPEL processes are exposed as Web services, and may themselves call other Web services. In this way, a BPEL process implements some business logic by orchestrating the use of external services.
### 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 async. 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 execution based on instance state.</td>
</tr>
<tr>
<td>pick</td>
<td>Conditional execution based on events.</td>
</tr>
<tr>
<td>flow</td>
<td>Concurrent execution.</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of BPEL Activities
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 2.1. Several vendors have implemented BPEL engines, including IBM, Microsoft, Oracle, and Sun Microsystems.

A business process is an aggregation of operations performed by people and software systems containing the information used in the process, along with the applicable business rules. The execution of a business process achieves a business objective. The people and systems may be inside the boundaries of a company, but often times are in multiple enterprises needing to collaborate to achieve their business objectives.

Business processes are executed by an *orchestration 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. Most existing BPEL engines support clustering in order to optimize and ensure business processes throughput on highly available systems. When a business process needs to be scaled to meet heavier processing needs, the BPEL engine’s clustering algorithm automatically distributes processing across multiple engines.

A comprehensive Business Process Management (BPM) platform provides an organization with the ability to collectively define their business processes, deploy those processes as applications accessible via the Web that are integrated with their existing soft-
ware systems, and then provide managers with the visibility to monitor, analyze, control and improve the execution of those processes in real time.

The BPM lifecycle can be classified into five categories: design, modeling, execution, monitoring, and optimization.

**Design:** Process design encompasses both the identification of existing processes and the design of “to-be” processes. Areas of focus include: representation of the process flow, the actors within it, alerts and notifications, escalations, standard operating procedures, service level agreements, and task hand-over mechanisms.

Good design reduces the number of problems over the lifetime of the process. Whether or not existing processes are considered, the aim of this step is to ensure that a correct and efficient theoretical design is prepared.

**Modeling:** Modeling takes the theoretical design and introduces combinations of variables, for instance, changes in the cost of materials or increased rent, that determine how the process might operate under different circumstances.

It also involves running “what-if analysis” on the processes: “What if I have 75% of resources to do the same task?” “What if I want to do the same job for 80% of the current cost?”

**Execution:** One of the ways to automate processes is to develop or purchase an application that executes the required steps of the process; however, in practice, these applications rarely execute all the steps of the process accurately or completely. Another approach is to use a combination of software and human intervention; however this approach is more complex, making the documentation process difficult.

As a response to these problems, software has been developed that enables the full business process (as developed in the process design phase) to be defined in a computer language which can be directly executed by the computer. The system will either use services in connected applications to perform business operations (e.g., calculating a repayment plan for a loan) or, when a step is too complex to automate, will ask for human
input. Compared to either of the previous approaches, directly executing a process defi-
nition can be more straightforward and therefore easier to improve. However, automating
a process definition requires flexible and comprehensive infrastructure, which typically
rules out implementing these systems in a legacy IT environment.

In this thesis, we present NIÑOS a completely distributed orchestration architecture
for business process execution based on the flexible PADRES publish/subscribe layer. It
executes business processes in an event-driven manner and is more in agreement with the
distributed nature of the processes themselves.

**Monitoring:** Monitoring encompasses the tracking of individual processes, so that
information on their state can be easily seen, and statistics on the performance of one or
more processes can be provided. An example of the tracking is being able to determine
the state of a customer order (e.g. ordered arrived, awaiting delivery, invoice paid) so
that problems in its operation can be identified and corrected.

In addition, this information can be used to work with customers and suppliers to
improve their connected processes. Examples of the statistics are the generation of mea-
sures on how quickly a customer order is processed or how many orders were processed
in the last month. These measures tend to fit into three categories: cycle time, defect
rate and productivity.

The degree of monitoring depends on what information the business wants to evalu-
ate and analyze and how business wants it to be monitored, in real-time or ad-hoc. The
level of monitoring depends on what event data is presented to humans. For example,
personnel managing running processes can see raw execution status data. Personnel re-
sponsible for service level agreements (SLA) see SLA status data produced by processing
raw event data. Executives see high-level business performance metrics. Here, busi-
ness activity monitoring (BAM) extends and expands the monitoring tools in generally
provided by BPMS.

Process mining is a collection of methods and tools related to process monitoring.
The aim of process mining is to analyze event logs extracted through process monitoring and to compare them with an ‘a priori’ process model. Process mining allows process analysts to detect discrepancies between the actual process execution and the a priori model as well as to analyze bottlenecks.

**Optimization:** Process optimization includes: retrieving process performance information from modeling or monitoring phase; identifying the potential or actual bottlenecks and the potential opportunities for cost savings or other improvements; and then, applying those enhancements in the design of the process. Overall, this creates greater business value.

To illustrate the application scenario, we present an example of real-world business process management deployment. 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.
Chapter 3

Related Work

3.1 Publish/Subscribe Systems

The publish/subscribe paradigm [13, 65, 22, 70, 27] provides a simple communication metaphor for entities that require complex interaction patterns while remaining decoupled. Data producers publish data to a broker which forwards the data to consumers who have subscribed to them. In this way, producers and consumers remain anonymous. The publish/subscribe implementations are differentiated by the expressiveness of the subscription language and the routing protocol. Most existing publish/subscribe systems are based on predicated-based language model and subscriptions are matched by individual publications. Subscribers are not allowed to correlate and aggregate publications using complex event patterns because of the limited expressiveness of the language. Another limitation of the traditional publish/subscribe model is that it only delivers to subscribers those publications produced after the subscription was issued. It is a model to query the future. Many content-based routing approaches [13, 65, 22, 70, 49] are based on an acyclic overlay broker network. Since there is only one path between any pair of brokers or clients in the overlay, the content-based routing protocol is greatly simplified. However, the flexibility to accommodate changing network conditions and
robustness with respect to broker failures are sacrificed. Different routing optimizations are explored in these systems. In this section we will list and compare some well-known publish/subscribe systems in terms of the expressive of their languages and the routing protocols.

### 3.1.1 SIENA

SIENA [13] is a distributed content-based publish/subscribe system consisting of an event broker network. The goal of SIENA is to provide an event notification service useful in a wide area network. The two primary services that it supports are: notification selection, which determines which components get which notifications, and notification delivery, which forwards the notifications to the correct components. SIENA focuses on the trade-off between scalability in notification delivery and expressiveness in the notification selection mechanism. Its subscription language is based on structured records, but allows for multiple predicates on the same attribute. A subscription is a conjunction of predicates. The semantics of its subscription language, however, do not include composite subscriptions, thereby omitting an entire category of expressiveness. The functionality of subscribing to historic publications is not considered in SIENA.

To minimize network traffic, it uses advertisements to avoid subscription flooding, and explores covering-based routing for subscriptions. Subscriptions in SIENA are represented in a partially ordered set (\( \text{poset} \)), and the partial order is defined by the covering relations. However, the \( \text{poset} \) is expensive to maintain because of the nested covering relations. SIENA supports general network topologies in the content-based routing protocol. However, the solution does not offer the potential to adaptively route based on network conditions. The routing paths setup for message dissemination are the shortest paths at the time that subscriptions are forwarded and can not adapt over time. Moreover, their solution discards duplicated messages, instead of avoiding them in the first place.
### 3.1.2 REBECA

REBECA [65] is another content-based publish/subscribe research system. REBECA provides a formal semantics and explores advanced content-based routing algorithms. The formal specification of publish/subscribe systems uses sequential traces and is based on the syntax of linear temporal logic so that it is easy to reason about the correctness of the routing algorithms. Advanced content-based routing algorithms are proposed including identity-based routing, and merging-based routing. Identity-based routing is a simplified version of covering-based routing. It presents two covering algorithms to detect covering relationships among subscriptions. One algorithm determines all subscriptions that cover a given subscription; the other algorithm determines all subscriptions that are covered by a given subscription. The two algorithms are based on the predicate counting algorithm. However, the algorithms suffer from the fact that they have to linearly scan the subscription sets twice. Merging-based routing is more advanced and exploits the concept of filter merging. However, REBECA only focuses on perfect merging. Imperfect merging as developed in our work is not discussed. All these routing protocols are based on an acyclic broker overlay.

REBECA [65] uses a predicate-based language model, similar to SIENA. Due to the expressiveness of the language itself, the composite subscription and historic data access function are not explored in REBECA.

### 3.1.3 Hermes

Hermes [75] is a content-based notification service implemented on top of a peer-to-peer overlay network. The prototype uses rendezvous nodes. A rendezvous node is responsible for a certain event type. A scalable routing algorithm using an overlay routing network is presented to avoid global broadcasts by creating rendezvous nodes. Hermes presents a type- and attribute-based publish/subscribe model, which is similar to the content-
based routing paradigm by first specifying the event type and then filtering within the event attributes. The abstraction of an overlay routing layer helps to hide some of the routing complexity and improves the scalability of Hermes. Hermes exploits covering-based routing. However, merging-based routing and composite subscriptions are not addressed.

3.1.4 Gryphon

Gryphon [70], a research project at IBM Research, is a distributed content-based messaging system. The techniques developed in Gryphon are complementary to the ones of SIENA. It focuses on defining an efficient algorithm to match publications against a large number of subscriptions. The algorithm exploits commonalities among subscriptions. In other words, whenever two or more subscriptions specify a constraint on the same attribute, the algorithm organizes them in order to evaluate the value of that attribute in each publication only once for all the subscriptions. Since the main focus of this project is on defining a fast algorithm for matching and limiting the network traffic concerning event delivery, Gryphon does not explore advertisement-based routing and covering-based routing so that subscriptions are propagated everywhere in the network. The issue of subscription routing cannot be disregarded since it can dramatically increase the network traffic in a wide area network.

3.1.5 Le Subscribe

Le Subscribe [24] aims at publish/subscribe support for Web-based applications. It focuses on the algorithmic efficiency in supporting millions of subscriptions and high event-processing rates. The language and data models are based on an LDAP-like semistructured data model for expressing subscriptions and publications. In this system, a subscription is a conjunction of predicates. This system supports both push- and pull-based information dissemination. The matching engine of Le Subscribe falls within the class of
two-step matching algorithms - a predicate matching step and a subscription evaluation step. In the first step, all predicates are matched against the publication. In the second step, subscriptions are evaluated based on the set of matched predicates.

### 3.1.6 PADRES

This thesis is based on the PADRES distributed content-based publish/subscribe system with features inspired by business process management systems. It adopts the advertisement-based routing from SIENA and REBECA system. PADRES consists of a set of brokers forming a peer-to-peer routing overlay. Each broker has a rule-based matching engine which performs message matching and routing. That is, publications are mapped into facts and subscriptions are mapped into rules. The matching between rules and facts is performed by a Rete [29] network. The rule-based approach naturally enables *composite subscriptions*. The subscription language in PADRES allows subscribers to describe event patterns. A subscriber is notified only when its event pattern is detected in the broker network. Event correlation is first addressed within publish/subscribe paradigm by PADRES. Another novel feature of PADRES is allowing the subscriber to access data published both in the past and in the future. So from the client’s point of view, it works as a distributed database. Databases are integrated into the publish/subscribe federation as a subscriber and a publisher for historic data.

PADRES also explores the covering-based and merging-based routing. The goal of covering- and merging-based routing are to minimize the number of messages routing in the broker network by eliminating redundant routing information at each broker. For example, if a subscription is covered by another existing one, the new subscription is not forwarded into the system, or if some subscriptions have highly overlapped publication set, they can be merged into a more general subscription which is forwarded into the system instead of the original ones, etc. It also supports robust content-based routing by allowing general broker overlays with cycles, unlike most existing publish/subscribe
systems [13, 65, 22, 70, 27] which are based on an acyclic overlay broker network. The cyclic overlay provides multiple paths from one client to another so that PADRES is robust to broker failures and message congestion, which are not avoidable in acyclic overlays. PADRES provides dynamic message routing and optimizes the composite subscription routing by taking advantages of the cyclic overlay.

We implement a distributed business process management system based on PADRES in which we can perform decentralized BPEL execution, business process modelling, business process deployment, and advanced functions for business process monitoring and control. We also implement a monitor module which is incorporated to graphically display the topology of the publish/subscribe overlay broker network and the messages routing between brokers. The prototype is to proof that decentralized business process execution is the trend for next generation products, and the publish/subscribe model is ideal to serve as a enterpriser service bus (ESB) for distributed applications.

3.2 Complex Event Processing

In traditional publish/subscribe systems, when a publication reaches a broker, the broker matches the publication against its databases of subscriptions and forwards the publication along the appropriate network links. Each publication match is an independent stateless operation. Composite subscriptions, which consist of primitive (atomic) subscriptions, are analogous to composite event detectors in event-processing systems. A composite subscription is matched only after all component atomic subscriptions are satisfied. That is, a composite event pattern in the publish/subscribe system occurs and is detected. Composite events are a key concept in the event-processing context and have not received much attention in the publish/subscribe literature.
3.2.1 SNOOP

Composite event detection first arose in the context of triggers in active databases. Languages for specifying composite events follow the Event-Condition-Action (ECA) model. The motivation for Snoop [18] is to design an expressive composite event specification language with powerful temporal support. An event specification language should have complete expressiveness and should be easily detected. Snoop is a model independent event specification language, which meets the increasing needs of complex applications and provides the two features mentioned above.

Composite events can be described using event expressions in Snoop. In a complex event expression, five kinds of operators are supported: disjunction, conjunction, sequence, aperiodic event operators, and periodic event operators. To make the interpretation of composite events precise, Chakravarthy et al. [18] introduce the notion of consumption policy, which is used to capture application semantics by resolving which events are consumed from the event history login for composite events in case of ambiguity. An efficient algorithm is given for detecting events expressed in Snoop. A composite event detector is a tree that reflects the structure of the event expression. Its nodes implement language operators and conform to a particular consumption policy. The detection propagates up the tree with the leaves of the tree being primitive event detectors. The event trees can be merged to form an event graph for detecting a set of composite events. The advantage of using the event graph is to avoid the detection of common sub-events multiple times thereby reducing storage requirements.

Although Snoop is much better than other event languages regarding expressiveness, extensibility, and efficiency, there are still several issues not fully addressed by the authors. First, for simplicity, Snoop assumes that events do not occur at the same time, while simultaneous occurrence of events is natural in a multiprocessor and distributed network environment. Second, the composite event detection is centralized, not distributed across a broker network. Furthermore, the algorithm can be optimized by considering the
equivalence of Snoop expressions. For example, two equivalent expressions do not need to be detected redundantly. Finally, the consumption policies are operator-dependent and nonintuitive.

### 3.2.2 GEM

GEM [60] uses a rule-based event monitoring language in which the notion of real time has been closely integrated and various temporal constraints can be specified for event composition. It assumes the existence of a global clock so that events occur in a total time order. Communication latency is handled by annotating rules with tolerable delays. This approach is not suitable for distributed environments with unpredictable delays. GEM uses a tree-based detection approach to process incoming events. In a basic GEM tree, atomic events are leaf nodes, and operators are inner nodes. The composite subscription is represented by the root of the tree. Each subscription in GEM has a tree, and duplicated sub-trees are not reused. GEM does not take advantage of common event patterns specified in different subscriptions. GEM focuses on a centralized approach for composite event detection.

### 3.2.3 Event Processing Service

Similar to GEM, EPS (Event Processing Service) [63] provides another tree-based event specification language. The atomic events are the leaf nodes. Each inner node may represent a composite subscription. That is, a subscription tree can be shared among subscriptions. EPS alleviates the main limitation of GEM by using a shared subscription tree to process incoming events.
3.2.4 Cambridge Event Architecture

CEA [77] presents a general composite event detection framework as an extension of an existing publish/subscribe middleware platform. The composite event detectors are automata. A core composite event language is used to express event patterns that occur concurrently. The language is compiled into automata for distributed detection, making the framework more scalable and robust. The composite event detection framework is distributed, and is strictly separated from the publish/subscribe infrastructure.

3.3 Stream Processing

A number of distributed applications involve information arriving continuously over time from distributed sources in the form of data streams, and this has led to new ways of processing queries in a widely-distributed environment. Stream-based distributed applications must cope with a critical issue: how to efficiently aggregate, use, and make sense of the enormous amounts of data. Centralized approaches to data aggregation suffer from high communication overheads, lack of scalability, and unpredictably high processing workloads at central servers. Multiple distinct middleware frameworks have been developed to support such data-intensive applications using distributed approaches. Exploiting knowledge of the underlying network characteristics for query processing and exploring the temporal dynamics of information streams for efficient query evaluation improve the overall performance, scalability, and availability of stream-based applications.

Continuous queries [82, 21, 59, 20] are issued once and run “continually” over the database. Tapestry [82] supports such queries over an append-only database via a limited SQL-like language, but does not support aggregations or joins. NiagaraCQ [21] uses an incremental query evaluation method but is not limited to append-only data sources. It uses a static query optimizer, and operators from different queries can be shared. CACQ [59] adds the notion of tuple-lineage to share queries beyond common query plan
Chapter 3. Related Work

subtrees, and uses the *eddy* operator to continuously adapt query workload, data delivery rates or system performance. PSoup [20] treats data and queries symmetrically as streams, allowing queries to access both data that arrived before the query registration and data in the future. However, PSoup uses a main-memory data structure limiting historic data to memory size. Also as with any centralized architecture, it presents a failure and performance bottleneck.

The notion of data streams is very similar to publications in publish/subscribe systems. A publish/subscribe system will continuously evaluate subscriptions over incoming publication streams until subscriptions are removed from the system. Composite subscriptions enable the correlation and aggregation of publication streams. However, content-based publish/subscribe differs from continuous queries and stream processing. First, a subscription may match publications from an anonymous and unknown number of data sources which may not conform to a predefined schema, whereas queries on streams typically explicitly identify the source streams and can rely on data conforming to known schemas, greatly simplifying query processing and routing decisions. Second, we allow clients to subscribe to both historic and future data. The historic data are stored in a distributed set of databases that are not limited by main memory. Third, none of the stream processing systems address data space partitioning and replication. We propose partitioning algorithms, define consistency properties that conform to publish/subscribe semantics, and develop protocols to maintain consistency among replicas in the publish/subscribe system.

Ahmad et. al. [4] proposed a novel network-aware approach for push-based continuous queries and distributed stream processing. The distributed query processing system is built on top of a Distributed Hash Table (DHT). Queries are represented as trees in which internal nodes are stream-oriented operators and leaf nodes are data sources. The authors described three algorithms for constructing query processing overlay networks: Edge, Edge+, and In-Network. Edge+ is a network-aware version of Edge, and In-Network
extends the first two by selecting query evaluation locations from a subset of the whole network instead of sources. The queries are evaluated based on the network topology, link bandwidth, and latency. The approach is experimentally evaluated in a prototype implementation and realistic network models. Comparison with representative network-unaware approaches verified that the proposed approach can significantly reduce the overall system bandwidth consumption and that the algorithms can be tuned to satisfy target query-result latency bounds.

Kleinberg [44] discussed approaches for the temporal properties of information streams. The goal is to exploit the essential part of the stream data, called episodes, and filter out the “noise” of information. Different approaches search for different types of patterns in information streams, organize episodes in a timeline, and rank them by significance. The approaches are threshold-based methods, state-based methods, and trend-based methods. Threshold-based methods detect episodes in which some features, such as the occurrence rate of key words, exceed the average rate by a specific threshold. State-based methods use probabilistic automata to detect episodes and they can filter out word bursts which cannot be handled by threshold-based methods. Trend-based methods are suitable for streams that have sets of words which exhibit the most pronounced rising and falling trends over the entire stream.

The above two projects discussed query processing from two different aspects: network awareness and temporal dynamics. Each of these approaches has its own limitations. The network-aware approach simplifies the query processing model by assuming that different stream operators have the same processing costs and selectivity, and only one data source matches a leaf node in a query. Moreover, it does not consider queries that are semantically equivalent so that the query processing network may not be optimal. The approaches for temporal dynamics focus on analyzing streams from one data source, limiting them to be a centralized version. Combining the two approaches, for example exploring temporal properties of streams as optimizations for distributed network-aware
query processing, makes it possible to further reduce network traffic and improve query evaluation performance.

Kumar *et al.* presented *IFLOW* [47, 46], an overlay messaging framework that provides the highly scalable and resource-aware approaches to distributed stream management. *IFLOW* is composed of three layers: the application layer, the middleware layer, and the underlay layer. The application layer is responsible for accepting and parsing the stream composition requests, which are described using an SQL-like language, and for constructing the data-flow graph. The middleware layer consists of the *Echo* middleware, which is a channel-based publish/subscribe system, and the proactive directory service *PDS*, which is a resource-monitoring infrastructure managing the network-wide resource availability information. The two components support the deployment and dynamic reconfiguration of the data-flow overlay. The underlay layer organizes the nodes into hierarchical partitions that are used by the deployment infrastructure.

The key problems addressed by *IFLOW* are reducing communication overheads by selectively streaming the events; distributing the processing workload by spreading the computing resources across the organization; and implementing easily used high-level language constructs for specifying new data-flow graphs. One of the important features of the infrastructure is the replaceability of components. For example, the *Echo* middleware, which is a channel-based publish/subscribe system, could be replaced by a content-based publish/subscribe system, such as *PADRES*.

Brokers in *PADRES* are different from distributed stream processing engines [47, 2, 19, 76] 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. These operators input and output a set of streams and may filter, or change the data on these streams. Borealis [2] is a distributed stream processing engine in which streams are queried by a network of operators. In addition to using a proprietary query language, Borealis does not support loops in the query network, which makes it unsuitable for business processes. In the *IFLOW* [47]
distributed stream processing engine, IFLOW nodes are organized in a cluster hierarchy, with nodes higher in the hierarchy assigned more responsibility. 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 our decentralized business process management system differs from the above work by exploiting an underlying content-based publish/subscribe system. Like [47], our agents are decoupled by communicating using publish/subscribe names instead of network identifiers. In addition, 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.

Query processing in multidatabase systems distributed over wide-area networks exhibits a number of complex performance problems. Selection of the best global query processing plan and dynamic optimization of data integration operations have the most important impact on performance. Query scrambling has been proposed to address these problems [7, 85, 31]. A key performance issue that arises in wide-area distributed information systems is response time unpredictability. Data access over wide-area networks involves a large number of remote data sources, intermediate sites, and communication links, all of which are vulnerable to congestion and failures. Such problems can cause
significate and unpredictable delays in the access of information from remote source. The objective of query scrambling is to change an order of operations in data integration expression such that the system is able to perform the other useful integration while it is waiting for the missing arguments.

Query scrambling [7] is heuristic-driven. It consists of two different phases: a rescheduling phase, in which the scheduling of the operators of an active query plan is changed when a delay is detected, and an operator synthesis phase in which the query plan is restructured, typically by creating new operators that are not in the current query plan. The heuristic rules may be effective at hiding delays in some situations, but they were also shown to be prone to making poor scrambling decisions in other cases. In some cases, the proposed heuristics could result in performance that is worse than simply waiting for the delayed data to arrive.

Graefe et al. [33] introduced choose-plan operators into a query plan to compensate for the lack of information about system parameters at compile time. Ioannidis et al. [42] generated multiple alternative plans, choose among them when the query is initialized. Neither of these approaches, however, can adapt to changes in the system parameters that occur during the query execution. Urhan et al. [85] extended their earlier work by addressing the shortcomings of the heuristic-based approach and proposing three different approaches to using query optimization for scrambling. They focused on the problem of initial delay which is the delay in receiving the first tuple from a particular remote source due to difficulty in establishing a connection to a remote source, heavy load of the remote source, and the large amount of work remote source needs to perform. They investigated both the use of response time-based and total work-based optimization for query scrambling. Getta [31] unified query scrambling with reduction approaches to dynamic optimize query processing plans at data integration stage. The objective of reduction is to eliminate from the arguments available at a central site all rows that have no impact on the final result of data integration. By unifying the query scrambling and
reductions, they removed the limitations of query scrambling to join expressions only and removed the limitation of reductions to compute one operation at a time.

We optimize the composite subscription routing in PADRES broker network. The composite subscription evaluation is similar to the query scrambling problem since they both retrieve data from multiple data sources and try to minimize the total work and delay for the results. The differences are the following. First, for composite subscriptions in publish/subscribe systems, part of the data may not be available and they may be published in the future. Second, publications may not follow the same schema as data in databases. As a result, the evaluation of composite subscription can not take this advantage as query scrambling did.

3.4 Distributed Workflows

WFMSs (Workflow Management Systems) support human beings in the execution of business processes, also called workflows or applications. Workflow management systems are traditionally centralized, creating a single point of failure and a scalability bottleneck. Because of their monolithic and centralized architecture, they cannot cope with the requirements that large-scale workflow applications pose. Interaction and integration of existing workflows are not feasible in current WFMSs either. Solutions for high scalability and workflow interaction across heterogeneous platforms are anticipated for next generation of WFMS products. Distributed workflow processing has been studied in the 1990s to address scalability, fault resilience, and enterprise-wide workflow management [5, 87, 68]. Compared to centralized approaches, distributed approaches for WFMS are more suitable for workflows and business processes across multiple organizations, providing better throughput, response time, and availability.

Hagen et al. [35] describe an approach for event-based communication for workflows cooperating in a consumer-producer relationship. In the terminology of the authors,
events are parameterized signals which are raised by a running job to inform other jobs about certain situations occurring during its execution. During workflow execution, a workflow manager maintains a dependency graph, recording which jobs have been triggered by which events. The manager routes the events to particular job executors according to the dependency graph, and collects feedback events. The centralized management degrades the scalability of workflow management system in terms of performance of workflow execution and network traffic and constitutes a single point of failure.

To reduce the workload of a central manager, Bauer et al. [10] propose an approach that decomposes workflows into parts which are controlled by different workflow managers/subnets. During the execution, several managers coordinate with each other to finish the whole application. Instead of using one central workflow manager, multiple managers are included. Communication costs and workloads of individual managers can be reduced. However, the workflow decomposition and manager coordination are not trivial. Splitting workflows and distributing each part to the “right” subnets may be expensive.

The publish/subscribe paradigm is a natural fit for workflow management, as the loosely-coupled nature of publishers and subscribers relieves the manager from maintaining client connection and capability information. A workflow management system, called OPSS, is implemented on top of the JEDI framework. The message routing is transparent to the management layer. However, since JEDI does not support composite subscriptions, a centralized state server storing the state of the enacting workflow is needed in OPSS to coordinate the workflow execution. The state server collects all job execution messages and decides which job should run next. Although OPSS relieves the manager from message routing, which is performed in the publish/subscribe layer automatically, it constitutes a single point failure by using a state server to control the workflow execution.

Muth et al. [68] proposed a solution for distributed workflow execution. They pre-
presented a scalable approach to enterprise-wide workflow management based on the distributed execution of state and activity charts. By exploiting the formal semantics of state and activity charts, they were able to develop an algorithm for converting a centralized workflow specification into an equivalent partitioned specification, which is suitable for distributed execution. Each partition is executed by a workflow engine server. To synchronize workflow execution among multiple servers, servers need to communicate the new system configuration of each partition to all other partitions during execution. The authors further investigated when and how the configuration needs to be communicated. The goal was to minimize synchronization costs in terms of the number of synchronization messages and in terms of their sizes, while guaranteeing a workflow execution equivalent to that achieved with a non-distributed approach.

In contrast, Casati et al. [16] focused on workflow interaction and integration. They presented a model for specifying interaction among workflows and a system that implements the model and provides the communication infrastructure. They extended traditional workflow models by allowing workflows to publish and subscribe to events, and by enabling the definition of points in the process execution where events should be sent or received. Event notifications are managed by a publish/subscribe model that filters, correlates, and dispatches them to the appropriate target workflow instances.

There are similarities and differences among the two projects. Muth et al. and Casati et al. both solved a similar problem, namely, improving the scalability of current workflow management systems. The first project tried to split a large-scale workflow into partitions and execute them distributively, and the second project tried to integrate small workflows into a large-scale workflow, which is also a solution for distributed execution. Furthermore, if we take the individual business processes as partitions of a larger workflow, Casati et al.'s approach uses a publish/subscribe messaging substrate to synchronize the distributed workflow execution. The publish/subscribe model greatly simplifies the synchronization, compared to Muth et al.'s algorithm, which is based on
a cost model. However, the event dispatching in Casati’s publish/subscribe model is preconfigured, and so is not flexible for large-scale applications. A fully content-based publish/subscribe model is more appropriate in this scenario.

Replicating workflow engines in a distributed environment is another natural approach to decentralizing the execution of workflows and achieving better quality of service (QoS). The critical issue is how to automatically configure a distributed WFMS such that it can guarantee QoS specifications at the application level.

Gillmann et al. [32] present a tool, Goliat, for the automatic configuration of a distributed workflow system that allows to meet specified goals for workflow performance and availability. Goliat’s core asset is a suite of analytic models which use continuous time Markov chains (CTMC) and Markov reward models to predict the behavior of workflow execution. The performance model estimates the throughput, response time and turnaround time while capturing the behavior of complicated workflow executions in a more realistic manner, for example, workflow instances with loops and arbitrary subworkflows. The authors also extended the CTMC model to support both geometric and uniform distributions for the number of loop iterations. This novel approach for deriving quantitative information from a distributed WFMS captures the behavior of a workflow system in a realistic manner and is easily adapted to existing workflow systems. However, it has two limitations which may restrict the wide use of the Goliat tool in commercial workflow systems. First, the configuration tool is based on online monitoring of workflow execution. Although the execution is distributed, the monitor may be a centralized bottleneck and a single point failure in a large-scale workflow system, which limits the scalability of the WFMS. Second, the authors ignore the heterogeneity of workflow servers by assuming that each server is well-configured and as powerful as the others. This assumption is highly unrealistic because WFMSs typically involve across multiple organizations and platforms. It is more reasonable to assign a greater number of workflow tasks to powerful servers and balance the workload according to the ability
of the servers.

Alonso et al. [5] presented a detailed design of a distributed workflow management system. In this work, a business process is fully distributed among a set of computing nodes. However, the distribution architectures differ fundamentally from PADRES. In PADRES, 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.

A behavior preserving transformation of a centralized activity chart, representing a workflow, into an equivalent partitioned one is described in [68] and realized in the MENTOR system [87]. The objective of the work is to enable the parallel execution of the partitioned flow, while minimizing synchronization messages, and analytically prove certain properties of the partitioned flow [68]. This is complementary to our work since we operate with the original business process model without analyzing the process.

The development of the business process execution language for Web services (BPEL4WS) is part of ongoing activities to standardize a family of technologies for Web services. It is a language designed to describe the behavior of business processes based on Web services. The 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.

There are several vendors who have implemented BPEL execution engines, including IBM, Microsoft, Oracle, and Sun Microsystems. These engines are centralized and must manage an entire BPEL process on one machine. Scalability is typically addressed by load balancing process instances to a cluster of engines, where each engine still executes the entire process. In PADRES 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. In order to executing BPEL in a distributed environment cross multiple BPEL engine agents, we need to understand the formal semantics of BPEL.

Researchers have proposed formal semantics for BPEL. There are existing attempts based on process algebras [26], a finite state machine [88], and Petri nets [37].

Ferrara [26] proposed a framework for the design and verification of Web services using process algebras and their tools. He defined a two-way mapping between abstract specifications written using the calculi and executable Web services written in BPEL. Process algebras are useful for temporal logic verification, service redundancy analysis, and replacement of one service with another in a composition. In the framework, the following choices are available: design and verification in BPEL using process algebra tools, or design and verification in process algebra together with automatically obtaining the corresponding BPEL code. The approaches can be combined.

Wombacher et al. [88] discussed the concept of matching business processes in loosely coupled architectures. In particular, a transformation from BPEL to annotated deterministic finite state automata (aDFA) was defined. The authors selected a finite state automata approach because most of the features in BPEL can be interpreted as message sequences representing a regular language. The aim of this paper was to provide a transformation from BPEL to aDFA notation, which is similar to transformations from regular expressions to finite state automata.

Ferrara [26] and Wombacher et al. [88] do not support some of BPEL’s most interesting features such as fault, compensation, and event handling. Hinz et al. [37] presented a Petri net semantics for BPEL. Their approach covers the standard behavior of BPEL as well as exceptional behaviors, e.g., faults, events, and compensations. The semantics is implemented as a parser that translates BPEL specifications into the input language of the Petri net model checking tool. They demonstrated that the semantics is well suited
Chapter 3. Related Work

for computer aided verification purposes. Their goal was to build a technology chain that, starting with a BPEL process, performs static analysis. Based on the analyzed information, the parser generates a Petri net model, and a model checker verifies the relevant Petri net properties. The verified Petri net model indicates that the BPEL process is feasible.

Among the three approaches, the Petri net approach supports complete BPEL features and has computer aided verification, in particular, model checking. It is useful for large-scale business processes. However, the models generated by the parser are significantly larger than manually generated models.

In PADRES [52, 27], we support the transformation of the complete set of BPEL features, including fault, compensation, and event handling. Each activity is processed by a BPEL activity agent which is a publish/subscribe client acting as both a subscriber and a publisher. Basically, an activity agent subscribes to its predecessor activity and publishes for its successor activity. The BPEL process is transformed into a set of publish/subscribe messages which can be understood by the PADRES system. The communication among agent is based on the publish/subscribe messaging substrate. After the transformation, the BPEL process in format of a set of publish/subscribe messages is deployed into the broker network and each activity is assigned to proper activity agents in publish/subscribe layer. In the execution phase, a trigger publication message is issued from the process manager to generate an execution instance. When an activity is finished, its agent publishes a publication indicating this event, the publication is received by the successor activities. The whole execution is event-driven until the last activity is finished. There is no centralized control once the execution starts. The execution is performed in the publish/subscribe layer as well. Because of the loosely coupled feature in publish/subscribe the business process monitoring and control are flexible. It allows multiple process instance running concurrently. It also possible to pause some of the instances to apply process online updates. To monitoring the execution, the manager
can subscribe to execution information at activity level and process level.
4.1 Subscription Language Features

The subscription language is used by subscribers to specify their interests. The language should provide several properties. First, the language should be powerful enough to describe expressive subscriptions. It should support not only basic relational operators, such as AND and OR, but also advanced features, such as variables, time sequence operators, and so on. Second, the language should be notationally simple. Subscribers must be able to write composite subscriptions easily and succinctly. That is, the syntax and semantics should be intuitive. Third, the language must have a mathematically precise concept of a match. That means when we specify a composite subscription, we know exactly which composite events will match it. Last, event patterns described by the language should be easy to detect. That is, event detection should be easy to implement.

There is a trade-off between the language expressiveness and detection efficiency. The more powerful the language is, the more complicated event pattern it can express, on the other hand, the more complex the composite event detection would be. Therefore, to design a subscription language, we need to understand the applications’ requirements, and provide a complete, but minimal operator set that is enough to describe the required
interests.

The PADRES subscription language is augmented for the support of composite subscriptions. Each composite subscription is comprised of several component subscriptions connected with logical relational operators such as and and or. Furthermore, composite subscriptions in PADRES can have variables bound to values in the publications. Variables are represented by $'s in subscription predicates. Braces {} are used to specify the priority of operators. A composite subscription is represented by a subscription tree, where the internal nodes are logical operators and leaf nodes are primitive subscriptions, as shown in Figure 4.1. A composite subscription is mapped to a complex rule in the PRT, and each component subscription is part of the rule. When a publication matching one of the component subscriptions arrives, the composite subscription is stored in a partially matched state. When all components are matched, the composite subscription is considered matched. For example, [class, eq, trigger], [appl, eq, payroll] \(\land\) [class, eq, job_status], [appl, eq, payroll] is satisfied after publications of both trigger and job_status classes, and of application payroll are published. Another composite subscription with variables [class, eq, trigger], [appl, eq, $x$] \(\land\) [class, eq, job_status], [appl, eq, $x$] is satisfied after publications of both trigger and job_status classes, of the same application, are published. The variable $x$ causes the join on application name. These features are sufficient to support distributed business process management as described in Chapter 8.

4.2 The Subscription Language

We extend publish/subscribe to allow subscribing to publications in the future and the past. As the former problem has been the focus of publish/subscribe and the latter addressed by relational databases, it is instructive to compare the two models.
4.2.1 Basic Operations

Fundamentally, both publish/subscribe and databases are concerned with getting data from producers to interested consumers, and support the following four operations.

**Define schema**: An SQL administrator issues a `CREATE TABLE` command to define a data template. Likewise, a publish/subscribe producer issues an *advertisement*.

**Produce data**: SQL data producers `INSERT` tuples into tables, and publish/subscribe producers *publish* publications.

**Query data**: SQL consumers query tuples with `SELECT` statements; publish/subscribe consumers *subscribe* to publications.

**Consume data**: The results of SQL `SELECT`s are returned to consumers as query results, whereas matching publications are delivered to publish/subscribe subscribers as *notifications*.

The principal difference in the above operations between database and publish/subscribe clients is that the order these operations are invoked: in the database model data is first produced, and queries return data produced in the past after which the query is complete, whereas publish/subscribe queries are evaluated continuously and return data produced after the query is issued. This thesis extends the publish/subscribe model to support querying data from the past.
We now point out some key differences between the relational database and publish/subscribe models.

**Data format**: While both models structure data as attribute-value pairs (or tuples), publish/subscribe publications are not relational. Moreover, publications may include only a subset of the attributes defined in the corresponding advertisement schema. This is equivalent to null values in database tuples.

**Data collections**: Database tuples, organized in tables, can be added (**INSERT**) or overwritten (**UPDATE**) by any producer. In publish/subscribe, data is only loosely organized as ordered publications corresponding to a particular advertisement. Furthermore, updates are not supported and the semantics are somewhat like an append only database.

**Query source**: While an SQL **SELECT** statement only queries those tables explicitly specified, a publish/subscribe subscription queries all publications.

**Query expressiveness**: Most publish/subscribe languages cannot express SQL notions of projections or joins.

### 4.2.2 Requirements

In our approach as described in Section 6.1, publish/subscribe clients query future data in the usual way, but retrieve historic data from a set of provisioned databases that serve as repositories of previously published data. This difference, however, should not be visible to clients and we seek a single language to query both historic and future data. This language should satisfy the following requirements.

**Retain publish/subscribe semantics**: The language should support the current content-based publish/subscribe model, including the predicate constraints and queries for future data.

**Future and historic queries**: There should be a unified way to query any combination of future or historic data.

**Simple mapping to SQL**: Since we use standard relational databases to store historic
data, it is desirable for queries to be easily mapped to SQL if querying historic data.

**Joins and projections**: The query language should have some limited ability to express joins and projections.

To avoid introducing an unfamiliar language, we restrict ourselves to extending either SQL or a publish/subscribe language. Since SQL is more expressive and is easy to “map” to SQL, we choose the former option, and call our language PADRES SQL (PSQL). We emphasize it is not our goal to develop a model that is a superset of databases and publish/subscribe, but to extend the publish/subscribe semantics with some concepts from SQL. For example, our data model is not relational, new data do not overwrite old ones, and queries do not support the full SQL feature set.

### 4.2.3 PSQL Language

The PSQL language is described using the framework of the four operations outlined above. To remind the reader that we are largely retaining publish/subscribe semantics, we will use publish/subscribe terms such as subscriptions and publications as opposed to queries and tuples.

The examples below will use a scenario where a country’s border security services monitors shipments into a country. Three sources of events are available: radioactive readings of shipments by sensors, shipping manifests issued by shipping firms, and internal audit reports on these firms.

**Define schema**: Each data producer (i.e., publisher) specifies a template that describes the publications it will publish. This is traditionally done with an advertisement message, but we adopt the equivalent SQL table creation statement.

\[
\text{CREATE TABLE (attr op val[, attr op val]*)}
\]

PSQL’s `CREATE TABLE` differs from that in SQL. Tables are unnamed since they need not be referred to by subscriptions or publications. Also, the range of values of each
attribute (or column) can be specified. Moreover, regardless of the attribute value constraint, each attribute can implicitly be a null value. The following statements setup the three event sources in our scenario. Sensor readings include the shipment id and a non-negative radioactivity level less than 10; shipping manifests indicate, for a given shipment, the expected contents of the shipment, and the shipping firm; and audit events indicate the trust level of a shipping firm.

\[
\text{CREATE TABLE (type = reading, shipID = *, level < 10)}
\]
\[
\text{CREATE TABLE (type = manifest, shipID = *, firm = *, content = *)}
\]
\[
\text{CREATE TABLE (type = audit, firm = *, trust >= 0)}
\]

In the above example, * is a wildcard that indicates the corresponding attribute may have any value.

**Produce data:** A publication is produced using a construct similar to SQL’s `INSERT` statement.

\[
\text{INSERT (attr[, attr]*) VALUES (val[, val]*)}
\]

The following publications are compatible with the advertisement schema defined above.

\[
\text{INSERT (type, shipID, level)}
\]
\[
\text{VALUES (reading, 123, 4)}
\]
\[
\text{INSERT (type, shipID, firm)}
\]
\[
\text{VALUES (manifest, 123, ACME)}
\]

Notice that only a subset of attributes defined in the schema need to be specified.

**Query data:** Subscribers issue `SELECT` statements to query both historic and future publications.

\[
\text{SELECT [ attr | function ]*}
\]
\[
\text{[FROM event]}
\]
\[
\text{WHERE [ attr op val]*}
\]
\[
\text{[GROUP BY attr*]}
\]
\[
\text{[HAVING function*]}
\]

The `FROM` and `HAVING` clauses are optional and are used to express joins and aggregations as described below.

A traditional publish/subscribe subscription for future publications would look as follows in PSQL.
SELECT *
WHERE type = reading, level > 3

Note that the above statement does not query a single table, so the results may have any number of attributes. The only guarantee is that all notifications will have the type and level attributes with values constrained as specified.

Subscribers can specify time constraints using reserved attributes start_time and end_time in the WHERE clause. Time constraints can be used to query for publications from the past, the future, or both. For example, upon a heightened security alert, it may be necessary to also retrieve suspicious sensor readings shortly before the alert was issued. The following subscription queries data in a time window that begins one hour before the time the query is issued and extends into the future.

```
SELECT *
WHERE type = reading, level > 3,
    start_time = NOW - 1h, end_time = NOW + 4h
```

The system internally splits the above subscription: one purely historic subscription that is evaluated once, and one ongoing future subscription. A subscription for both historic and future data is a hybrid subscription.

Composite subscriptions [52] can be expressed with simple join conditions. The event correlation is supported using the FROM clause, where the event pattern can be specified using Boolean expressions. To avoid false alarms, the subscription below (which is both hybrid and composite) will only report on shipments that are detected as radioactive, but whose material is not expected to be so. Furthermore, manifests are only trusted if the firm has been audited as trusted anytime from two months ago.

```
SELECT *
FROM e1 AND e2 AND e3
WHERE e1.type = reading, e1.level > 3,
    e2.type = manifest, e2.content != fertilizer,
    e3.type = audit, e3.trust > 7,
    e3.start_time = NOW - 2 months,
    e1.shipID = e2.shipID, e2.firm = e3.firm
```

The event in the FROM clause specify that three different publications are required to satisfy this query, and each publication must qualify the WHERE constraints. The three
publications may come from different publishers, and may conform to different schema.

Notice that a composite subscription can collect, correlate, and filter publications in the network. Without this feature, a user must retrieve sensor readings for all potentially dangerous sensor shipments, and then issue a historic query for the associated manifest, and then another query for the associated audit record. This would be expensive (both for the user and in terms of network traffic) in cases where the sensor events are generated frequently.

Event aggregation is supported in PSQL as well. The \texttt{HAVING} clause can specify constraints across a set of matching publications. The functions $\texttt{AVG}(a_i,N)$, $\texttt{MAX}(a_i,N)$, and $\texttt{MIN}(a_i,N)$ compute the appropriate aggregation across attribute $a_i$ in a window of $N$ matching publications. The window may either slide over matching publications, or be reset when the \texttt{HAVING} constraints are satisfied. For example, perhaps because sensors are faulty, the following subscription will only match when the average reading from ten different sensors indicate radioactive material.

\begin{verbatim}
SELECT *
WHERE type = reading
HAVING AVG(level, 10) > 3
GROUP BY shipID
\end{verbatim}

Any attributes specified by functions in the \texttt{HAVING} clause must appear in the publication. So, an implicit \texttt{level = *} condition is added to the \texttt{WHERE} clause above. Also, the \texttt{GROUP BY} clause has the same semantics as in SQL and serves to constrain the set of publications over which the \texttt{HAVING} clause operates.

\textbf{Consume data}: Notification semantics do not constrain notification results, but transform them. PSQL supports projections and aggregations over matching publications to simplify notifications delivered to subscribers and reduce overhead by eliminating unnecessary information.

Projections are a useful feature rarely supported in publish/subscribe. Notifications may include a subset of attributes in matching publications with the \texttt{SELECT} clause.

\begin{verbatim}
SELECT shipID, level
WHERE type = reading, level > 3
\end{verbatim}
The above subscription only guarantees that matching publications include the *type* and *level* attributes. Notifications may contain only a subset of the *shipID* and *level* attributes.

Aggregation functions may appear in the **SELECT** clause. For example, the following subscription returns the average sensor reading for all shipments.

```sql
SELECT shipID, AVG(level, 10)
WHERE type = reading
GROUP BY shipID
```

Again, an implicit *level = * condition is added to the **WHERE** clause since the *level* attribute appears in a function.

Note that functions in the **SELECT** and **HAVING** clauses have different semantics. In the preceding subscription, for every ten matching publications, a single notification is delivered to the subscriber containing the average radioactive level, whereas the subscription with the **HAVING** clause we saw earlier delivers all publications within a window of ten whose average satisfies the specified condition.

### 4.2.4 Discussion

The PSQL language unifies access to publications from the past and the future. It retains predicate-based publish/subscribe subscription functionality, and provides a consistent way to query any combination of future and historic publications with simple conditions on the *start_time* and *end_time* attributes. Evaluating future PSQL subscriptions is straightforward, and mapping historic subscriptions into SQL statements is simplified by basing the language on SQL. An SQL-like language also afforded a familiar construct to express simple joins (composite subscriptions), and notification semantics such as projections and aggregation constraints, which are not typically supported by publish/subscribe systems.
Chapter 5

Robust Content-based Routing

Applications on top of the publish/subscribe layer require robust message delivery services. That means, when failures happen in the system, messages can be delivered in the best efforts, unless no routing paths is available, i.e., network is partitioned by failures. Most existing publish/subscribe systems [28, 64, 70] are based on an acyclic overlay broker network. With only one path between any pair of brokers or clients, content-based routing is greatly simplified. Despite this success, an acyclic overlay offers limited flexibility to accommodate changing network conditions, is not robust with respect to broker failures, and introduces complexities for supporting other protocols, such as failure recovery. For example, since only one path exists between any pair of clients, an acyclic overlay cannot accommodate routing around congested, overloaded, or failed brokers. Furthermore, because acyclic networks are more vulnerable to partitions, failure recovery is more expensive and topology reconfiguration can be complex since their repair actions must maintain the acyclic property of the overlay [74]. Maintaining the acyclic property is difficult since a broker often only knows about its direct neighbors and not the entire topology.

To make this work self-contained, we discuss composite subscription routing and then present the content-based routing protocol in general overlays.
5.1 Composite Subscription Routing

Subscribers may require a higher level view of publications in a publish/subscribe system, and may wish to subscribe to multiple publications which vary in type and time of occurrence. This kind of subscription is referred to as a *composite subscription*. A *composite subscription* consists of a set of primitive subscriptions connected by logical operators, which are supported by the subscription language. A *composite subscription* describes a complex event pattern that occurs in the publish/subscribe system. The constituent publications that combine to satisfy a composite subscription are often termed *atomic publications* or *atomic events*, since publications can be taken as events. When a composite subscription is satisfied by a set of publications, a composite event pattern is detected. Thus we call the process of matching a composite subscription *composite event detection*. There are two kinds of composite event detection: centralized and distributed. A centralized event server is needed in centralized approach. All events are routed to the server, and the composite event pattern is detected by the server. Distributed detection is more flexible. A composite subscription is split into parts, and the detection of each part is close to the corresponding event sources. The results of detection could be shared among subscribers. If a composite subscription is matched, a *notification* is sent back to the subscriber, instead of a set of component publications.

Composite subscriptions allow subscribers to express their subscriptions in composite event patterns. Subscribers are relieved from dealing with a large number of atomic events. Composite subscriptions make the publish/subscribe system easy and efficient to use in terms of expressiveness of the subscription language and network bandwidth. Furthermore, the feature of supporting composite subscriptions is the key to modeling applications like business process management systems in the publish/subscribe paradigm.

In a large-scale publish/subscribe system, publications are issued at geographically dispersed sites. A centralized composite event detection schema could form a bottleneck and cause a single point of failure, as all the primitive publications have to be collected
in order to detect an occurrence of a composite event. We propose a mechanism for
distributed composite event detection. The main difficulties are to decide where and how
to decompose a composite subscription, and how to route the individual component sub-
scriptions. Firstly, the location of detection should be as close to publishers as possible.
This is because the publications contributing to a certain composite subscription would
not be unnecessarily disseminated throughout the broker network. In other words, the
composite subscription should be forwarded within the broker network as far as possible
before it is decomposed. As a result, bandwidth usage is reduced. Second, the decom-
position should occur along the routing path of a composite subscription. Since the
matching publications may come from different brokers, a composite subscription should
be split into parts and each part should be routed separately. Each part is routed to
its corresponding publishers. Later, matching publications would come to the broker at
which the composite subscription is split, and the detection is performed at that broker.
In addition, the detection results should be shared among subscribers that have common
subexpressions of composite subscriptions to save bandwidth and computational effort.

We suggest two algorithms for performing composite subscription routing and com-
posite event detection.

5.1.1 Simple Routing

We can detect the whole event pattern at the first broker that receives the composite
subscription from the client. The composite subscription needs to be fully factored
into primitive subscriptions, and forwarded into the broker network from there. All
the matching publications would be sent back to this broker, and the composite event
detection is performed there. A notification is sent to the client directly by the broker if
a composite subscription is matched. That is, the client directly connected to the broker
receives the notification as soon as the composite event pattern is detected. However,
collecting all potential matching publications at the first broker increases network traffic
overhead. In a word, although clients have less detection latency for their subscriptions, this is not an ideal solution in terms of the use of network resources and bandwidth.

### 5.1.2 Topology-based Routing

The second option is to perform a distributed composite event detection in the network. To do this, the routing of composite subscriptions is more complicated than for primitive subscriptions. The main difficulty when routing composite subscriptions is to decide the optimal placement to decompose them within the system and route the primitive subscriptions to the correct brokers. Basically, if all component subscriptions have the same next-hop destination, a broker would forward the composite subscription as a whole to the destination; otherwise the composite subscription should be split into parts according to different destinations, and each part should be forwarded to its own destination separately. The routing scheme is used to detect the event pattern matching a composite subscription at a location which is as close as possible to the data sources. A composite subscription is mapped to a rule, and a publication is mapped to a fact in the matching engine. The rule-based matching engine matches facts against rules and decides where to route the notification if there is a match. The matching and routing are the same as primitive subscriptions in PADRES. It acts as both a message router and an event detector. The advantage of using a rule-based matching engine is that it can support composite subscriptions without significant changes to the engine itself.

There are several advantages of using distributed composite event detection. Redundant detection is eliminated by sharing the detection results among subscribers. For the overlapping expressions of composite subscriptions issued by clients, the detection is executed once, and subscribers close to each other can reuse the detection results. Distributed detection also reduces network traffic. A composite subscription is forwarded into the network as far as possible before it is split. As a result, the number of subscriptions injected into the network does not increase rapidly because of the composite
subscriptions. Furthermore, composite events are detected close to data sources in the network. A single notification is sent after a match, instead of a set of publications, reducing the number of publications routed in the federation.

<table>
<thead>
<tr>
<th>buildDestinationTree(cs):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> composite subscription cs</td>
</tr>
<tr>
<td><strong>Output:</strong> a destination tree T</td>
</tr>
<tr>
<td>1: Initialize T according to cs</td>
</tr>
</tbody>
</table>
| 2: If (cs.root is leaf node) {
| 3: T.destination = cs.root.destination |
| 4: } |
| 4: Else{
| 5: T.left = buildDestinationTree(cs.root.left) |
| 6: T.right = buildDestinationTree(cs.root.right) |
| 7: If T.left.destination == T.right.destination {
| 8: T = T.left.destination |
| 9: } |
| 9: Else{
| 10: T =null |
| 11: } |
| 11: } |
| 12: Return T |

Figure 5.1: Algorithm of Building a Destination Tree

When a broker receives a composite subscription, three routing steps are performed:

1. A destination tree is built bottom-up for the composite subscription according to the SRT in the matching engine, which knows where all the component subscriptions come from. Leaf nodes are destinations of component subscriptions; an internal node is the destination of its child nodes if the two child nodes have the same destination, or null otherwise. If a node is null, all its parent nodes are null. The recursive algorithm for building such a tree is presented in Figure 5.1. Suppose \( N \) is the number of primitive subscriptions in a composite subscription. The average time complexity of this algorithm is \( O(N) \). The average space complexity is \( O(N + \log N) \), where \( N \) is for working space array and \( \log N \) is for the stack space.

2. The composite subscription is split according to its destination tree. Each node in the composite subscription tree has a corresponding node in the destination tree.
If a node’s destination is *null*, the subscription represented by the node is split into two parts, one for each child node. If a node’s destination is not *null*, the node and its subtree will be routed as a single unit. The decomposition process is top-down. The algorithm is given in Figure 5.2. The time and space complexity of this algorithm is the same as algorithm \textit{buildDestinationTree}(cs).

3. All the component subscriptions are routed to their destinations as specified by the destination tree.

$$\text{decomposition}(cs, T):$$

\begin{verbatim}
Input: composite subscription cs; destination tree T
Output: a set of component subscriptions S
1: Initialize S = 0
2: If (T.destination is null){
3: \hspace{1cm} S = S \cup \text{decomposition}(cs.root.left, T.left) \cup \text{decomposition}(cs.root.right, T.right)
4: }
4: Else{
5: \hspace{1cm} S = S \cup cs
6: }
7: Return S
\end{verbatim}

Figure 5.2: Algorithm of Decomposing a Composite Subscription

Figure 5.3 shows a simple example. Suppose a composite subscription \(cs: \{\{s_1 \text{ and } s_2\} \text{ or } s_3\}\) comes from Broker 1, and its matching publications come from different brokers, as shown in the left hand side of Figure 5.3. At Broker 1, since all matching publications are coming from Broker 2, \(cs\) is routed as a whole to Broker 2. At Broker 2, the SRT shows that publications matching \(s_1\) and \(s_2\) will come from Broker 4, while \(s_3\)'s publications will come from Broker 3. A destination tree is built, and \(cs\) is split into two parts: \(\{s_1 \text{ and } s_2\}\) and \(s_3\). \(\{s_1 \text{ and } s_2\}\) is sent to Broker 4, where it is split into \(s_1\) and \(s_2\), and sent to Broker 5 and Broker 6, separately. \(s_3\) is a primitive subscription, and it is routed to Broker 3.

A more complex subscription routing example is given in Figure 5.4. Suppose \(cs\) comes from Broker 1. It is forwarded as a single unit in the network until it reaches Broker 11. At Broker 11, as shown in the left hand side of Figure 5.4, the composite
subscription is split into three parts that would be routed to Broker 6, Broker 10, and Broker 12 separately according to the destination tree. Among the three parts, \( s_4 \) and \( s_5 \) are already primitive subscriptions, therefore they are routed as regular subscriptions to their destinations. While \( cs' = \{(s_1 \text{ or } s_2) \text{ and } s_3\} \), which is routed to Broker 10, is a composite subscription. It needs further decomposition. Similarly, the decomposition happens at Broker 10 and Broker 4. Thus, each component subscription is routed to its corresponding publishers.

5.1.3 Unsubscription of Composite Subscriptions

Unsubscription Protocol

In PADRES, if a client wants to unsubscribe a primitive subscription, the client issues an unsubscription message. The process of unsubscription is not trivial. Since a subscription is distributed on different brokers within the network, the unsubscription should be an atomic transaction in order to maintain the consistency of routing tables. We propose an unsubscription protocol with \( ack \) messages. An \( ack \) message is sent if a broker removes
Chapter 5. Robust Content-based Routing

a subscription from its matching engine successfully. The protocol has three steps:

1. When a broker receives an unsubscription, it checks the SRT to find the list of
neighbor brokers to which it previously routed the subscription.

2. If the list is empty, it removes the subscription from its routing table, and sends
back an ack message. Otherwise, it forwards the unsubscription to all the brokers
in the list, and waits for ack messages from them.

3. The broker cannot safely delete the subscription until it collects all the ack messages
back from its neighbors. An ack message is sent back to the broker/client who
forwards the unsubscription.

The ack messages and the protocol ensure the atomicity of unsubscription process.
Without the protocol, it would be necessary to wait for a certain time interval for brokers
to process the unsubscription message. However, the consistency of routing tables within
the broker network is not guaranteed. Each ack message is distinguished by a message id,
so the unsubscription of several subscriptions can be executed concurrently. The unsub-
scription of a composite subscription is more complex, since its component subscriptions

Figure 5.4: A Complicated Composite Subscription Routing Example
are widely distributed in the network, and it is necessary to unsubscribe all of them properly in order to maintain the consistency of SRTs and PRTs in the broker network. Next, we explain how to perform the unsubscriptions for composite subscriptions.

**Decomposition at the First Broker**

```plaintext
unSubFirstBroker(unSub);
Input: unsubscription unSub
Output: ACK message ack

1: If(unSub is an unsubscription of a composite subscription cs) {
3:   Set S = cs’s primitive subscription set
4:   For each subscription si in S {
5:     Create unsubscription message for si
6:     Compute the routing list listi of si
7:     If listi is null {
8:       Remove si from SRT and PRT
9:       Send acki for si
10:   }
11:   Else{
12:     Forward si to all the brokers in listi
13:   }
14: }
15: }
16: If (unSub is an unsubscription of a primitive subscription sub) {
17:   Create unsubscription message for sub
18:   Compute the routing list of sub
19:   If the list is null {
20:     Remove sub from SRT and PRT
21:   }
22:   Else{
23:     Forward sub to all the brokers in the list
24:   }
27: }
25: Wait for all acks from neighbor brokers
26: Create an ack message ackUnSub for unSub
28: Return ackUnSub
```

Figure 5.5: Algorithm of Unsubscribing Subscriptions at the First Broker

If a composite subscription is split at the first broker when it is routed, the unsubscribe is easy to perform. A client’s unsubscription message is first received by the broker to which the client is connected. At the first broker, the unsubscription message is fully decomposed into a set of primitive unsubscriptions. These unsubscription messages are forwarded to the next-hop brokers. Each unsubscription message of the component
subscriptions is processed as a primitive unsubscription in PADRES. The first broker could delete the composite subscription after it receives all the ack messages from its neighbors. An ack message is sent back to the client by the broker if the process of unsubscribing is successful. Figure 5.5 shows the unsubscription algorithm. Suppose $N$ is the number of primitive subscriptions in a composite subscription. The average time complexity of this algorithm is $O(N)$.

Figure 5.6 shows a simple example. Suppose $cs: \{\{s_1\} \text{ and } \{s_2\}\}$ was issued by a client from Broker 1. To unsubscribe it, the client issues an $UnSub(cs)$ message. When Broker 1 receives the unsubscription message, it generates two unsubscriptions: $UnSub(s_1)$ and $UnSub(s_2)$, and sends the two messages to Broker 2. For each unsubscription message, say $UnSub(s_1)$, Broker 2 continues to forward it to Broker 4, to which $s_1$ was previously routed. After it receives the ack message from Broker 4, Broker 2 deletes $s_1$ from its matching engine, and sends an ack message indicating that $s_1$ has been removed from the system back to Broker 1. The same thing happened to $UnSub(s_2)$. When Broker 1 receives both $ack(s_1)$ and $ack(s_2)$, it can safely remove composite subscription $\{\{s_1\} \text{ and } \{s_2\}\}$ from its matching engine. Thus, the composite subscription is unsubscribed correctly.

**Distributed Decomposition**

Figure 5.7 describes the algorithm of unsubscribing a distributedly decomposed subscription. The unsubscription in the distributed case is similar to the first case. However,
unSubDistributed(unSub):
Input: unsubscription unSub
Output: ACK message ack

1: If(unSub is an unsubscription of a composite subscription cs ) {
2: Destination Tree $T =$ buildDestinationTree(cs)
3: Set $S =$ decomposition(cs, T)
4: For each subscription $s_i$ in S {
5: Create unsubscription message for $s_i$
6: Compute the routing list $list_i$ of $s_i$
7: If $list_i$ is null {
8: Remove $s_i$ from SRT and PRT
9: Send $ack_i$ for $s_i$
10: }
11: Else{
12: Forward $s_i$ to all the brokers in $list_i$
13: }
14: }
15: }
16: If (unSub is an unsubscription of a primitive subscription $sub$ ) {
17: Create unsubscription message for $sub$
18: Compute the routing list of $sub$
19: If the list is null {
20: Remove $sub$ from SRT and PRT
21: }
22: Else{
23: Forward $sub$ to all the brokers in the list
24: }
25: }
26: Wait for all $acks$ from neighbor brokers
27: Create an $ack$ message $ackUnSub$ for $unSub$
28: Return $ackUnSub$

Figure 5.7: Algorithm of Unsubscribing Distributedly Decomposed Subscriptions
the composite subscription is not fully decomposed at the first broker, but split according to its destination tree as discussed in Section 5.1.2. The component subscription set $S$ in $unSubFirstBroker(unSub)$ consists of primitive subscriptions, while $S$ in $unSubDistributed(unSub)$ includes both primitive subscriptions and composite subscriptions, depending on the way $cs$ is decomposed according to the destination tree. The complexity of this algorithm is $O(N + 2\log N)$.

Figure 5.8 shows how to unsubscribe the previously routed composite subscription illustrated in Figure 5.3. Suppose $cs$’s subscriber issues an unsubscription message for $cs$. At Broker 1, before $cs$ is removed from the SRT and PRT, the unsubscription message is forwarded to Broker 2, and Broker 1 waits for an $ack$ message. Similar to the routing of $cs$, Broker 2 also splits the unsubscription into two parts, and it sends $UnSub(s_1 \text{ and } s_2)$ to Broker 4 and $UnSub(s_3)$ to Broker 3. The same thing happens at Broker 4. After Broker 5 and Broker 6 receive the unsubscription messages of $s_1$ and $s_2$, and remove them from their SRTs and PRTs respectively, $ack$ messages are sent back to Broker 4. Broker 4 can safely remove $\{s_1 \text{ and } s_2\}$ after it collects all the $ack$ messages. So does Broker 2. Finally, Broker 1 receives the $ack$ message from Broker 2, and removes $cs$ from its SRT and PRT. The $ack$ messages guarantee an atomic unsubscription process. Each $ack$ message is distinguished by a unique message id, so the unsubscription of several

![Figure 5.8: Unsubscription of Distributedly Routed Composite Subscriptions](image-url)
composite subscriptions can be performed concurrently.

## 5.2 Composite Event Detection

![Composite Subscription Matching](image)

A composite subscription is routed into the broker network. When publications matching part of the composite subscription are issued, the publications would trace back along the path built by the composite subscription. In contrast to primitive publication matching, if several publications match a composite subscription, it is not a set of publications but a notification indicating the occurrence of a composite event pattern that is forwarded to the subscriber. A notification message is a publication as well, with a set of matching publications as its payload. The occurrence of a composite event is marked by the occurrence of the last event that makes the composite event occur. Since publications of the same kind may be issued periodically, different possibilities for detecting composite events exist. The rule-based matching engine acts as a composite event detector. The main problem for composite event detection is how to consume the publication events received by the brokers. To illustrate this, consider the composite subscription \{\{s_1 and s_2\} and \{s_3\}\}, where \(s_i\) matches publication type \(e_i\), \(i=1 \sim 3\). Each publication type has several instances, for example \(e_1\) has \(e_{11}\) and \(e_{12}\), as shown in Figure 5.9. We introduce the following strategies for event consumption in composite event detection:
**Recent** In this context, only the most recent occurrence of each $e_i$ is taken into account for the occurrence of a composite event. For instance, $<e_{12}, e_{22}, e_{33}>$ is the publication set that satisfies the composite subscription.

**Chronicle** In this context, publications are consumed in the chronological order in which they are published for composite event detection. In this case, $<e_{11}, e_{21}, e_{31}>$ is the correct solution for the composite event detection.

**Unlimited** This context gives all the possible combination patterns that satisfy the composite event. Clients are allowed to filter the results to get what they want.

In Padres, subscriptions are mapped to rules and publication are mapped to facts. Composite subscriptions in Padres turn out to be more complex rule expressions in the rule-based matching engine. Our strategy for composite subscription matching is a combination of unlimited context and a constraint that at least one of the publications in the pattern is issued after the composite subscription. There are two reasons to chose this strategy. First, we assume that subscribers in publish/subscribe systems are interested in future data most of the time. The occurrence of the composite event should be later than the time the composite subscription is issued. Subscribers could retrieve historic data through the historic data access method in Padres. Second, we try to satisfy the requirements of different application scenarios by providing a more complete set of composite event detection results. Different applications can easily use a filter for the results in order to receive the right patterns they are actually interested in.

Figure 5.10 shows the Rete network of the previous example $\{s_1 \text{ and } s_2\} \text{ and } \{s_3\}$. The initial Rete network has five events that occur before the subscription. If we now add a new fact $e_{32}$ to the Rete, four possible composite event patterns are detected. Each has the new fact $e_{32}$. The patterns are only detected once. When more new events come later, the discovered patterns would not be repeatedly detected. Applications are allowed to customize the result filter according to their application semantics. After the filtering, a notification carrying the result is built and sent to the subscribers.
Figure 5.10: Adding a Fact $e_{32}$ to Rete Network
5.3 Adaptive Routing in General Overlays

5.3.1 Challenges

Figure 5.11: Cyclic Routing of Subscriptions

Cycles in the network introduce misleading message routing paths and redundant traffic. In the network in Figure 5.11(a), advertisements $Adv_1$ and $Adv_2$ are broadcast and form two advertisement trees. Since brokers discard duplicate advertisements, the trees shown in Figure 5.11(a) may vary based on relative message delays. Now, a subscription $Sub_1$ matching both $Adv_1$ and $Adv_2$ arriving at Broker 1 from Broker X is forwarded both towards Broker 6 along the $Adv_1$ tree, and towards $Adv_2$. Unfortunately, at Broker 6, $Sub_1$ matches $Adv_2$ and is routed back to Broker 1, forming a cycle.

Another scenario of a subscription cycle is shown in Figure 5.11(b), where Broker 4, forwards $Sub_2$ to Brokers 6 and 2, following the paths built by $Adv_1$ and $Adv_2$, respectively. However instead of stopping at Brokers 5 and 1, the two copies of $Sub_2$ continue to be routed unnecessarily. The duplicate subscriptions are not detected until they arrive at the same broker, say Broker 3.

Subscription cycles not only cause redundant subscription messages, they can cause publications to be routed indefinitely in cycles as they keep switching among matching
In summary, cycles in the overlay lead to duplicate advertisements, subscriptions or publications, a problem that is exacerbated in well-connected networks with many redundant paths. However, since there are relatively few advertisements, detecting and discarding advertisements is advantageous, provided subsequent cyclic routing of subscriptions and publications can be prevented.
5.3.2 TID-based routing

We describe extensions to the standard content-based routing using Figure 5.12(a) as a running example. These extensions are contained within the routing protocol and do not modify the interface to the publish/subscribe clients.

**Advertisement:** Each advertisement is assigned a unique tree identifier (TID) within the broker network. In our implementation we use message identifiers, which are unique in our system, as TIDs.

Normally, when a broker receives an advertisement, it broadcasts the advertisement to its neighbors and inserts the advertisement in its subscription routing table (SRT). For cyclic networks, we extend this behavior such that brokers discard duplicate advertisements upon receipt, so each advertisement forms a spanning advertisement tree distinguished by TIDs. As we will see, our approach only requires such duplicate detection for advertisements, which we expect to have fewer of than subscriptions and publications.

**Subscription:** When a broker receives a subscription from a subscriber, it adds an existential predicate \([\text{TID} = \text{Z}]\) that uses a variable binding mechanism. If the subscription matches an advertisement in the SRT, the advertisement’s TID is bound to the variable \(\text{Z}\). For example, in Figure 5.12(a), \(\text{Sub}_1\) is matched by both \(\text{Adv}_1\) and \(\text{Adv}_2\) at Broker 1. A copy of \(\text{Sub}_1\) with TID bound to \(\text{Adv}_1\) is forwarded to Broker 6 and another copy bound to \(\text{Adv}_2\) is forwarded toward the publisher. In our implementation, the TID attribute may have a set of values associated with it so that only one copy of the subscription is forwarded if subscriptions with different TIDs have the same next hop.

A subscription with a bound TID value only propagates along the corresponding advertisement tree. Therefore, when a broker receives a subscription with a bound TID value, it can forward the subscription without matching the subscription against all the

\[\text{Predicate tests whether a message contains a TID attribute, in which case the value is bound to the variable in the subscription. Otherwise, the predicate is false.}\]
advertisements in the SRT. As a result, subscription forwarding is greatly sped up by the use of TIDs, by matching subscriptions against advertisements only once.

Subscriptions set up paths for routing publications. We extend the publication routing table (PRT) to a list of \([\text{subscription}, \{\text{TID, last hop of subscription}\}]\), such as \(PRT_1\) in Figure 5.12(b). Since a subscription may arrive at a broker via different paths labeled by TIDs, the PRT records the TID and the last hop broker of the incoming subscription. A subscription not in the PRT is inserted; otherwise the existing record is updated with the new \(\{\text{TID, last hop of subscription}\}\) pair.

![Figure 5.13: Multiple Publication Routing Paths](image)

A subscription matching multiple advertisements may be bound to several TIDs, and will form alternate routing paths for publications if the subscription with different TIDs has different last hops. For instance in Figure 5.13, \(Sub\) matches both \(Adv_1\) and \(Adv_2\) at Broker 1, and is assigned two TIDs in \(PRT_1\) with different last hops. Copies of subscriptions with different TIDs propagate along their corresponding advertisement trees and these paths may diverge and reconverge at a broker due to intersections among
the advertisement trees. Thus, a broker may receive multiple copies of a subscription with different TIDs. These are not, however, duplicate messages as they correspond to different paths, and are stored in the PRT as potential routing path alternatives for publications.

Alternative paths for publication routing are maintained in PRTs as subscription routing paths with different TIDs and destinations. More alternate paths are available if publishers’ advertisement spaces overlap or subscribers are interested in similar publications, which is often the case for many applications with long-tailed workloads. Our approach takes advantage of this and uses multiple paths available at the subscription level.

Subscription covering, merging, and summarization optimizations \[13, 49, 64, 84\] eliminate redundant subscriptions and result in smaller routing tables. These optimizations can be applied among subscriptions with the same TID.

**Publication:** When a broker receives a publication from a publisher, the publication is assigned an identifier equal to the TID of its matching advertisement. From this point, the publication is propagated along the paths set up by matching subscriptions with the same TID without matching the content of the publication at each broker. This simple and fast *fixed publication routing* algorithm is enabled by the use of TIDs. Alternatively, the *dynamic publication routing* algorithm described in Section 5.3.3 exploits alternate paths.

Notice that with the TID extensions, subscriptions form a directed, cycle-free graph between publishers and their potentially interested subscribers, so publications are never forwarded in a cyclic manner. In the directed graph, there may be multiple paths between any pair of brokers depending on how subscriptions are routed along multiple advertisement trees. In fixed publication routing, brokers do not need to detect duplicate publications and, consequently, no bandwidth is wasted due to redundant publication traffic.
**Property 1:** No broker receives duplicate publication messages.

*Proof.* By way of contradiction, assume publication $p(c, t)$ with content $c$ and TID $t$ is received by some broker $B_i$ twice.

Let broker $B_0$ be the broker that receives a publication $p(c, t)$ from the publisher. Without loss of generality, let $B_i$ be the first broker to receive $p(c, t)$ twice. Notice that $B_i$ must have received $p(c, t)$ from two different brokers $B_m$ and $B_n$, since brokers never forward the same publication multiple times to a neighbor, denoted as $B_m \xrightarrow{p(c, t)} B_i$ and $B_n \xrightarrow{p(c, t)} B_i$.

Observe that if $p(c, t)$ is forwarded over link $B_m \rightarrow B_i$, then a subscription $s$ with a TID bound to $t$ was sent over link $B_i \rightarrow B_m$, denoted as $B_i \xrightarrow{s_i} B_m$. And if such a subscription was sent, then an advertisement $adv$ with a TID $t$ was sent over link $B_m \rightarrow B_i$, that is $B_m \xrightarrow{adv} B_i$. Similarly, we have $B_n \xrightarrow{adv} B_i$. The spanning advertisement tree rooted at $B_0$ must have two paths from $B_0$ to $B_i$: $B_0 \rightarrow B_m \rightarrow B_i$ and $B_0 \rightarrow B_n \rightarrow B_i$.

This is a contradiction since trees only have one path between any two nodes. □

It follows from Property 5.3.2 that no subscriber receives duplicate publications, since a subscriber connects to exactly one broker, brokers forward a publication at most once over a link, and no broker receives duplicate publications.

### 5.3.3 Dynamic Publication Routing

Subscriptions are routed to publishers along advertisement trees. If the advertisement trees of different data sources intersect, multiple publication routing paths to a subscriber result. In Figure 5.13, $Sub$ is forwarded to Broker 1 over two paths (Path 1 via Brokers 5, 3 and 1, and Path 2 via Brokers 5, 3, 4, 2 and 1). Publications of $Adv_1$ take the path through Broker 3, while publications of $Adv_2$ take the path through Broker 2 in the fixed routing approach. However, if the TID of a publication could be adapted in transit, a better path may be chosen. A publication of $Adv_2$ arriving at Broker 1, could be routed
to Broker 3 by changing its TID to Adv1 instead of being routed to Broker 2.

A routing algorithm is required to determine the “best” path, based on metrics such as the fewest hops or the shortest end to end delay. While similar to the routing problem in IP networks, those solutions cannot be applied to publish/subscribe systems directly. In address-based routing, the shortest path can be calculated based on a global topology graph, such as with link state routing [61], whereas our brokers are only aware of their overlay neighbors, a property we wish to retain for scalability and manageability. More importantly, IP networks can rely on the clustering of addresses; all nodes in a part of the network may have the same network mask, and can be represented by a single routing table entry. In content-based pub/sub, however, nodes’ addresses (i.e., their subscription), may not be clustered. This makes global optimizations infeasible, and instead we use a decentralized solution based on local link status information.

In the dynamic publication routing (DPR) algorithm, a broker forwards a message through the link with minimal cost, using the heuristic that this link is also on the minimum cost path to the destination. To dynamically select paths while network traffic loads or topologies change, each broker maintains a Link Status Table (LST). For example, to minimize the delay cost, the LST can store the link utilization ratio of each neighbor, and update the ratio whenever messages are sent or received over the link. The link utilization ratio is $U = \frac{r_{output}}{r_{sending}}$, where $r_{output}$ is the rate of messages entering the output queue corresponding to the link, and $r_{sending}$ is the rate of messages sent on that link. The link utilization ratio captures the queueing delay of a link. Other costs can also be modeled in the LST but are not considered in this thesis.

When a broker receives a publication, for each matching subscription that may come from multiple links with different TIDs, it selects the link with minimal latency, and assigns the corresponding TID to the publication. The algorithm ensures that, for one subscription with different TIDs, each representing a path from a publisher to the subscriber, only one publication is forwarded to one of the potential neighbors. Also, only
one copy of the publication is forwarded to a neighbor if several matching subscriptions come from the same last hop. For example, in Figure 5.13 \( PRT_1 \) shows that publications matching \( Sub \) have two available paths, through Broker 2 or 3. Consulting the LST, the broker will forward the publication to the destination with minimal delay, say Broker 3.

When a broker fails or a link is overloaded, its neighbors will detect the failure or congestion as a result of messages queueing up in the corresponding output queue, which cause the link utilization ratio to increase. Consequently, publications will be routed around the failed or congested broker. While this approach tries to use as many available paths as possible to route messages around failures and congestions, it cannot guarantee delivery, such as in the case of a network partition, until the failures have been repaired by a separate module.

Modifying a publication \( p \)'s TID in transit seems to change the set of subscribers notified of this publication, but this is not the case. Intuitively, the algorithm works because for any given subscription \( s_i \) from a subscriber \( S \) that matches \( p \), \( p \)'s TID is only changed to an advertisement’s TID that also matches \( s_i \). That means \( p \) will be delivered to \( S \) by “borrowing” branches of another advertisement tree. The DPR algorithm is formalized in Algorithm 1 and its correctness is proven in Property 5.3.3.

**Property 2:** Changing a publication \( p \)'s TID while in transit will not change the set of notified subscribers \( N \).

*Proof.* Consider a publication \( p(c, Adv_1) \) with content \( c \) and TID \( Adv_1 \), that is changed to \( p'(c, Adv_2) \) where \( Adv_1 \neq Adv_2 \). We know that any publication in the set \( P(c, t) \) where any \( Adv_x \in t \) intersects \( s_i \) from subscriber \( S \) and \( c \) matches \( s_i \) will be delivered to \( S \). The proof consists of two claims.

**Claim 1:** No subscriber \( S \notin N \) receives \( p' \). Note that \( S \notin N \) implies that \( S \) is not interested in \( p \), that is, it does not have a subscription \( s_i \) that matches \( p \).

By way of contradiction, assume an \( S \notin N \) receives \( p' \). Consider the Broker \( X \) that sent \( p' \) to \( S \). According to Algorithm 1, this means that Broker \( X \) found an \( s_i \) in its PRT
Algorithm 1 Dynamic Publication Routing

Require: An incoming publication \( p(c, TID_p) \)

Ensure: \( \text{forwardMsgs}: \) A set of publications with destinations and updated TIDs

1: \( \text{forwardMsgs} = \emptyset \)
2: \( S = \{ s_i | p \text{ matches } s_i \text{ in the PRT} \} \)
3: for \( s_i \in S \) do
4: \( \varphi = \{ [TID_j, \text{LastHop}_j] | [s_i, TID_j, \text{LastHop}_j] \in \text{PRT} \} \)
5: Find \( m \) such that link utilization ratio of destination \( \text{LastHop}_m \) is minimal in \( \varphi \)
6: \( \text{nextHop} = \text{LastHop}_m \)
7: if \( p'.\text{content} \neq \text{content} \) and \( p'.\text{nextHop} \neq \text{nextHop}, p' \in \text{forwardMsgs} \) then
8: \( p.\text{TID} = TID_m \)
9: \( p.\text{nextHop} = \text{nextHop} \)
10: \( \text{forwardMsgs} = \text{forwardMsgs}.\text{add}(p) \)
11: end if
12: end for
13: return \( \text{forwardMsgs} \)

whose last hop is \( S \) and which matches \( p' \). Therefore, subscriber \( S \) sent a subscription \( s_i \) that matched publication \( p' \). Since \( p \) and \( p' \) have the same publication content, \( s_i \) also matches \( p \), meaning that \( S \) is interested in \( p \) and \( S \in N \), which is a contradiction.

**Claim 2:** All subscribers \( S \in N \) receive \( p' \). Again, note that \( S \in N \) implies that \( S \) is interested in \( p \), and has a subscription \( s_i \) that matches \( p \).

By way of contradiction, assume an \( S \in N \) does not receive \( p' \). Therefore, it must be that \( p'(c, Adv_2) \notin P(c, t) \), in particular, \( Adv_2 \notin t \). But we know that \( p(c, Adv_1) \in P(c, t) \), and according to Algorithm 1, \( p(c, Adv_1) \) is changed to \( p'(c, Adv_2) \) only if both \( Adv_1 \) and \( Adv_2 \) intersect \( s_i \), implying that \( \{Adv_1, Adv_2\} \in t \). Then it must be that \( p'(c, Adv_2) \in P(c, t) \). This is a contradiction.

Together, the above two claims prove that the set of notified subscriber \( N \) does not change when a publication’s TID is changed in transit.

Our solution exhibits useful properties. First, it retains the publish/subscribe client interface. No changes to the publish/subscribe matching algorithm are required, since TID attributes are matched just like any other attributes. Second, with the TID attribute, optimizations can be performed at each broker to speed up and simplify sub-
scription and publication propagation. For example, subscriptions are matched only once while forwarded to publishers. Third, our approach generates duplicate messages only when broadcasting advertisements; subscription and publication forwarding do not create redundant traffic. Fourth, subscriptions may determine multiple routing paths for publications. The DPR algorithm can route publications around failed or congested brokers, making the system more robust to broker failures and dynamic network traffic. Moreover, the DPR algorithm selects efficient routes based on network conditions to minimize notification delay. This is useful in applications with quality of service constraints.

5.3.4 Evaluation

In this section, we experimentally evaluate our routing protocols in general overlay topologies. For publications, we use stock quote traces obtained from Yahoo! Finance. Lacking real subscription traces, we generate subscriptions for the stock quote publication with predicates following a Zipf distribution in order to model locality among subscribers.\(^2\) We explore the properties of the routing protocols by deploying a broker overlay in a controlled 20 machine local network consisting of machines with 4GB of memory and 1.86GHz CPUs. Unless otherwise stated, we evaluate the protocols in a 30 broker overlay with an average connection degree \((D)\) of 4. Each node is a complete content-based publish/subscribe broker that implements the protocols described in Section 5.3. As well, 20 publishers and 30 subscribers join the system within the first 30 seconds of the experiment.

We compare the end to end notification delay of the fixed (Section 5.3.2) and dynamic (Section 5.3.3) algorithms and measure the CPU and memory usage per broker. To demonstrate robustness and scalability, we also evaluate broker networks deployed on PlanetLab.

**End to End Notification Delay** The notification delay is computed as the average

\(^2\)The datasets used in the experiments are available at [1].
Figure 5.14: End to End Notification Delay in Dense vs. Sparser Topologies
Figure 5.15: End to End Notification Delay with More Publishers
Figure 5.16: End to End Notification Delay on PlanetLab
time between publishing at the publisher to the corresponding notification at the subscriber. This delay varies with the number of hops from the publisher to subscriber and the workload in the broker network, the latter of which depends on factors such as the overall publication rate and the number of subscriptions per broker. The delay metric includes the queueing delay, processing and matching time, and transmission delay at each hop.

When the publishers and subscribers join the system, advertisements are broadcast and subscriptions are matched with advertisements and forwarded towards potential publishers. This initial traffic contributes to network congestion and large notification delays, and is visible in Figure 5.14(a) where the end to end delay is plotted (in log scale) for each notification. After initialization, the delays of both dynamic and fixed routing algorithms stabilize, with the dynamic algorithm producing a 20.3% shorter average delay than fixed routing. Our results show that brokers in the overlay have different traffic loads. In the fixed routing case, the output queueing delay of an overloaded broker constitutes 25% of the total end to end routing delay. The dynamic algorithm is able to detect such congestion and balance publication forwarding across the overlay.

In a less connected network, the benefit of dynamic routing is diminished due to a lack of alternate paths. In Figure 5.14(b), with $D = 2$, the dynamic routing delay is only about 4.7% better than that of fixed routing. At some points in the experiment, the dynamic approach even performs worse than the fixed one because of the overhead of the path selection algorithm. While Figure 5.14(a) and Figure 5.14(b) demonstrate that the dynamic approach benefits from a well connected overlay, the benefit is not proportional to the connection degree. When the experiment is repeated with a fully connected topology, the delay with the dynamic approach is 4.1% worse than with the fixed approach. In this case, the overhead of maintaining link information does not offset gains in alternate path selection since all publishers and subscribers are already one hop from one another.
We observe that an increase in the publication rate causes the fixed routing approach to suffer worse notification delays. For instance, in Figure 5.15(a) when the publication rate is increased to 2400 msg/min, the fixed algorithm becomes overloaded with messages queueing up at brokers along the routing path, whereas the dynamic routing algorithm continues to operate by offloading the high workload across alternate paths. The results suggest that dynamic routing is more stable and can handle heavier workloads, especially in a well connected network. Incidentally, the small but periodic decreases in delay are an artifact of the periodic publication workload. The publications near the end of the workload match fewer subscribers and are filtered by our routing algorithms before they propagate far into the network. This results in less congestion in the network and faster routing of the remaining publications. This phenomenon is apparent in all of the experiments that use this workload.

In the second scenario, we increase the number of publishers to 40, and their advertisements overlap each other. As described in Section 5.3.2, overlapping advertisements increase the number of potential alternate paths. Figure 5.15(b) shows that the delay with the dynamic algorithm stabilizes 5 minutes sooner than with the fixed algorithm. Furthermore, the end to end delay with dynamic routing is 46.5% less than that with fixed routing. (We note again the log scale of the delay axis in Figure 5.14 ~ Figure 5.16). This compares with a relative benefit of 20.3% in Figure 5.14(a) where there were fewer publishers and hence fewer alternate paths.

The third scenario evaluates a dynamic workload. A new publisher now joins the system five minutes after the other clients, whose publication rates range from 400~800msg/min, and publishes a burst of 1500 msg/min for two minutes. We further remove a publisher after the burst. As illustrated in Figure 5.16(a), the dynamic approach stabilizes first as in previous experiments. While both algorithms suffer from increasing delays due to the burst, the dynamic algorithm maintains a smaller delay, is able to recover faster after the burst, and has a smaller steady state delay. Overall, the dynamic approach is more
resilient to dynamic workloads.

<table>
<thead>
<tr>
<th>Distance</th>
<th>Fixed (ms)</th>
<th>Dynamic (ms)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 hops</td>
<td>47.2</td>
<td>47.6</td>
<td>-0.8%</td>
</tr>
<tr>
<td>10 hops</td>
<td>64.5</td>
<td>52.9</td>
<td>18.0%</td>
</tr>
<tr>
<td>12 hops</td>
<td>74.4</td>
<td>60.6</td>
<td>18.6%</td>
</tr>
<tr>
<td>Max diff</td>
<td>57.7%</td>
<td>27.4%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Effect of Subscriber Distance on Delay

In Table 5.1, we list the average end to end delay experienced by three subscribers that are 6, 10, and 12 hops away from the publisher. When the publisher and subscriber are close to one another, the dynamic algorithm has fewer opportunities to find suitable alternate paths. In Table 5.1, with a 6 hop path length, there is even a 0.8% performance degradation resulting from the overhead of the dynamic algorithm. When the distance increases to 12 hops, the improvement is up to 18.6%. We also observe, reading down the columns in Table 5.1, that the delay between the 6 and 12 hop subscribers with the fixed approach is 57.7%, while the corresponding difference with the dynamic algorithm is only 27.4%. This suggests that the dynamic approach is less sensitive not only to publication traffic but also to the path lengths between subscribers and publishers.

An important observation from the above results is that our publish/subscribe routing algorithms actually benefit from a cyclic overlay by reducing the notification delay and increasing the resilience to loads. To demonstrate the robustness and scalability of our approach, we repeat our experiments on PlanetLab with a 50 broker overlay network. The heterogeneous and shared PlanetLab nodes and network make it difficult to derive repeatable and reliable results, but our evaluations on PlanetLab support the conclusions made from the controlled environment experiments. Figure 5.16(b) confirms that the dynamic algorithm stabilizes faster than the fixed one, and has a smaller notification delay.
**Faster Matching**  Both the fixed and dynamic routing algorithms have the potential to dramatically improve routing and matching performance. It takes our matching engine about 4.5 ms to match a publication against over 200,000 subscriptions per hop [52]. Once the TID attribute is bound (see Section 5.3.2), subsequent brokers only need to match the TID attribute instead of the full publication content. This can provide significant savings, especially with complex subscriptions, large workloads, or long path lengths. For the dynamic algorithm in the experiment associated with Figure 5.14(a), 1926 publications issued by publishers resulted in 16997 publication messages, requiring 16997 matching operations by brokers. With TID routing, where only the first broker performs full matching, only \( \frac{1926}{16997} = 11.3\% \) of the matching operations are required.

**Overhead of Dynamic Publication Routing**  We measure the CPU and memory usage for all 30 brokers in the network over time for the experiment shown in Figure 5.14(a). In the dynamic approach, we need to periodically (10 ms in this experiment) update a link status table at each broker. When a publication arrives, the broker selects a better path based on the link status table. The average CPU and memory usage per broker in the dynamic routing approach is 6.3% and 8.9% higher than those of the fixed routing approach, respectively. The busiest broker in the dynamic approach has the most neighbors and consumes up to 83.7% of the CPU processing capacity, while the other brokers only consume 16.6% of the CPU. The results show that fixed and dynamic publication routing consume similar CPU and memory usage. Therefore, the dynamic algorithm reduces the notification delay and the resilience to publication workloads without consuming much more system resources.

**Dynamic Routing with Failures**  In this experiment, we connect 20 publishers and 30 subscribers to a fully connected 30 broker network. We simulate broker failures by randomly killing some of the brokers, and measure the end to end delay of the dynamic routing approach with failures. The number of failures the system can tolerate depends on the connectivity of the network, and the position of the broker in the overlay.
For example, a failure that partitions the network will render it impossible to deliver publications across the partitions.

![Figure 5.17: End to End Notification Delay with Failures](image)

In Figure 5.17, immediately after the first broker failure, which occurs after about the 1000<sup>th</sup> notification, the notification delay increases by up to 89.1%. Routing around the failed broker temporarily introduces congestion at some other broker, but the dynamic routing algorithm detects the congestion and automatically balances the traffic among the remaining alternate paths. When the second broker failure occurs at around the 7000<sup>th</sup> notification, fewer alternate paths are available, and the notification delay now increases up to 217.5%. This time, however, the notification delay stabilizes at about 23.1% higher than before the failure, because the alternate paths the algorithm finds are longer. This experiment shows that the dynamic routing approach can route messages around failures and balance traffic among alternative paths.
Chapter 6

Historical Data Access

6.1 Architecture and Routing

We propose an enhanced publish/subscribe paradigm that allows subscribers to access both future and historic data with a single interface as described in Section 4.2. The system architecture, shown in Figure 6.1, consists of a traditional distributed publish/subscribe overlay network of brokers and clients, to which a set of databases are added. The databases are provisioned to sink a specified subset of publications, and to later respond to queries.

Subscriptions for future publications are routed and handled as usual [15, 22, 70]. To

![Figure 6.1: Architecture of Historic Data Access](image-url)
support historic subscriptions, databases are attached to a subset of brokers as shown in Figure 6.1. The set of possible publications, as determined by the advertisements in the system, is partitioned and these partitions assigned to the databases. A partition may be assigned to multiple databases to achieve replication, and multiple partitions may be assigned to the same database if database consolidation is desired. Partition assignments can be modified at any time, and any replicas will synchronize among themselves. The only constraint is that each partition be assigned to at least one database so no publications are lost. Partitioning algorithms as well and partition selection and assignment policies are described in Section 6.2 and evaluated in Section 6.3.

Subscriptions can be atomic expressing constraints on single publications, or composite expressing correlation constraints over multiple publications. This section discusses how atomic subscriptions retrieve, filter, and correlate combinations of historic and future publications from multiple sources; the routing of composite subscriptions are presented in Chapter 7

**Atomic Subscription Routing**

When a broker receives an atomic subscription, it checks the start_time and end_time attributes. A future subscription is forwarded to potential publishers using standard publish/subscribe routing \[15, 22, 70\]. A hybrid subscription is split into future and historic parts, with the historic subscription routed to potential databases as described next.

For historic subscriptions, a broker determines the set of advertisements that overlap the subscription, and for each partition \(R_i\), selects the database with the minimum routing delay. The subscription is forwarded to only one database per partition to avoid duplicate results. When a database receives a historic subscription, it evaluates it as a database query, and publishes the results as publications to be routed back to the subscriber. Upon receiving an END publication after the final result, the subscriber’s host broker unsubscribes the historic subscription. This broker also unsubscribes future subscriptions
whose end_time has expired.

6.2 Database Partitioning

We propose a new publish/subscribe model where databases provisioned as part of the broker network store publications and respond to historic subscriptions. The databases are integrated such that subscribers access both future and historic data with a single interface as described in Section 4.2. Databases can store different sets of publications for load balancing or administrative purposes, and databases can be setup as replicas for availability or fault tolerance. We address publication space partitioning in Section 6.2.1, and the management of database replicas in Section 6.2.2.

6.2.1 Publication Space Partitions

Partitioning the publication space allows us to provision one or more databases to store publications belonging to the partition and respond to queries for them. To simplify the processing of historic subscriptions, we only consider horizontal partitioning here, and reserve vertical partitioning for future work.

**Horizontal Partitioning:** An advertisement (i.e., the CREATE TABLE schema definition) describes the publication space of a publisher. The global publication space $\varphi(A) = \bigcup \varphi(A_i)$, where $A_i$ is an advertisement and $\varphi(A_i)$ is the publication set induced by $A_i$, can be divided into partitions, which are the smallest units maintained by a database. Publication space partitioning in the publish/subscribe model differs from database partitions in that we partition a global space instead of individual tables. Furthermore, we need not consider referential integrity constraints. A formal definition of a partition is given below.

**Definition** *Publication Space Partition*
A partition is a division of the publication space $\varphi(A)$ into distinct publication sets, denoted as $R = \sigma_F(\varphi(A))$, where $F$ is a selection formula in form of predicate conjunctions.

That is, a partition is a projection of the publication space on selection formula $F$, where publications in partition $R$ satisfy the constraints in $F$. A partitioning of the publication space should be complete and disjoint as defined below.

Completeness: Every publication $p$ belongs to at least one partition $R_i$. Formally, $\forall p \in \varphi(A), \exists i$ such that $p \in R_i$.

Disjointness: The publication sets of partitions do not overlap. Formally, $\forall p \in R_i, p \notin R_j, i \neq j$.

The partitioning algorithm takes as input the set of advertisements in the system, and a set of predicates observed in query workloads. First, the algorithm computes a set of simple predicates that will form the selection formulas. An iterative algorithm such as $\text{COM\_MIN}$ [72] can generate a complete and minimal set of predicates $P'$ given a set of predicates $P$. The publish/subscribe definitions of completeness and minimality differ, and we adjust the $\text{COM\_MIN}$ algorithm accordingly. A predicate set is complete if it covers all attributes and values defined in the publication space, and is minimal if predicates on the same attribute do not overlap. Since publications may have any subset of attributes defined in the advertisement, and there is no primary key concept, every attribute must be used to partition the publication space to guarantee completeness. Predicates are grouped according to attributes. Predicates across groups are orthogonal and predicates within groups are minimal.

Next, the algorithm derives a set of selection formulas, which are conjunctions of orthogonal predicates, as candidates for partitioning. Finally unnecessary selection formulas are eliminated. For example, for selection formula $f$, if $\varphi(A) \land \varphi(f) = \emptyset$, then

---

1In distributed databases, a set of predicates is complete if there is an equal probability of access by every application to any tuple belonging to any partition.

2Every predicate is relevant in determining a partition.
partition $R_f$ is empty, no publications match $f$, and we need not consider $f$ during partitioning.

The resulting partitions are guaranteed to be complete and disjoint because the algorithm uses a set of complete and minimal predicates.

**Uniform Partition Accessibility**: To balance load, a uniform distribution of partition access by historic subscriptions is desirable. To achieve this, we estimate the access probability $\rho(R_i)$ of a partition $R_i$ based on queries observed over a period of time $T$ as

$$\rho(R_i) = \frac{\sum s_j \rho(R_i(s_j)) \cdot |\sigma_{s_j \varphi(A)}|}{|\varphi(A)|},$$

where subscription $s_j$ is issued within $T$, and the access probability of $R_i$ by subscription $s$ is $\rho(R_i(s)) = \frac{|\sigma_{f \varphi(s)}|}{|\varphi(A)|}$.

Computing $\rho(R_i(s))$ requires two assumptions: independence of predicates in the selection formula $f$ and subset proportionality conservation. The former is denoted as

$$\rho(R_{p_1 \land p_2}(s)) = \rho(R_{p_1}(s)) \cdot \rho(R_{p_2}(s)) \tag{6.1}$$

and the latter assumption is represented as

$$\frac{\rho(R_{p_1 \land p_2}(s))}{\rho(R_{p_1 \land \neg p_2}(s))} = \frac{|\sigma_{p_1 \land p_2 \varphi(A)}|}{|\sigma_{p_1 \land \neg p_2 \varphi(A)}|} \tag{6.2}$$

By algebraic substitution and transformation of the above two equations, we obtain

$$\rho(R_{p_1 \land p_2}(s)) = \rho(R_{p_1}(s)) \cdot \frac{|\sigma_{p_1 \land p_2 \varphi(A)}|}{|\sigma_{p_1 \varphi(A)}|} = \rho(R_{p_1}(s)) \cdot \rho(R_{p_2} / R_{p_1}),$$

where $\rho(a/b)$ is the conditional probability of event $a$ given event $b$.

Algorithm 2 computes partitions such that the standard deviation of partition access probabilities is less than a given $\delta$. Until the standard deviation falls below $\delta$, the algorithm splits the partition with highest accessibility along the attribute with the maximum number of domain values.

**Example**: We demonstrate the partitioning algorithm with the previous example. To simplify the example, we use a subset of attributes to describe the publication space.

```
CREATE TABLE (shipID = *, level >= 0)
CREATE TABLE (shipID = *, firm = *, content = *)
```

Suppose we discover the complete and minimal set of predicates\(^3\) from a subscription

\(^3\)The *null* predicate value means the attribute is not present in a matching publication.
Algorithm 2 Uniform Partition Accessibility

Require: $F$: selection formulas, $\delta$: accessibility deviation

Ensure: Standard deviation of partition access probabilities $< \delta$

1: $R \leftarrow \{R_i | \sigma_{f_i \nu(h)}, f_i \in F\}$
2: repeat
3: $p \leftarrow \frac{\sum_{i=1}^{|R|} \rho(R_i)}{|R|}$
4: $\sigma \leftarrow \sqrt{\frac{\sum_{i=1}^{|R|} (\rho(R_i) - p)^2}{|R|}}$
5: if $\sigma > \delta$ then
6: $R_s \leftarrow \text{Max}(\rho(R_i), i = 1 \sim |R|)$
7: Select an attribute with $\text{Max}(|\prod_a R_i|)$ and $p_a$, so that $f_i' = f_i \land p_a$ and $f_i'' = f_i \land \neg p_a$
8: $F \leftarrow F - f_i + f_i' + f_i''$
9: $R \leftarrow R - R_i + R_i' + R_i''$
10: end if
11: until $\sigma \leq \delta$
12: return $R$

(workload, as shown in Figure 6.2. Choosing one predicate per attribute, there are $2 \times 3 \times 2 \times 2 = 24$ selection formula candidates, including those in Figure 6.2.

We eliminate candidates using implication rules derived from advertisements. One rule is to eliminate selection formulas that do not follow any advertisement schema. For example, $R_{f_1}$ is not defined by any advertisements and can be removed. Some partitions may be empty because no publication in these partitions will be produced. For example, since all readings have $\text{level} \leq 10$, $R_{f_5}$ is empty, and $f_5$ can also be eliminated from the candidate set. Hence, the publication space is divided into six partitions based on the remaining selection formulas. The access probability of the partitions are calculated.
based on observed workloads and the model discussed above. If partitions are not evenly accessed, a large partition, say $R_{f4}$, can be further split by, for example, predicates $(level < 5)$ and $(level \geq 5)$.

**Discussion:** The granularity of partitions may range from a single partition for the global space to one per possible publication. Increasing partition granularity presents tradeoffs with the overhead of managing more partitions, potential parallelization of partition access, and performance overhead of routing the corresponding advertisement issued by the database. The appropriate number of partitions is application-specific and can be tuned with different sets of predicates input to the partitioning algorithm, or with an appropriate choice of $\delta$.

Partitions can not only be distributed among databases, but may be replicated across them. Replicas improve the fault-tolerance of the system, and provide an opportunity to select the closest replica to answer a historic subscription. However, having to maintain consistency among the replicas increases the complexity of the system.

### 6.2.2 Database Provisioning

Subscriptions for future publications are routed and handled as usual by the publish/subscribe broker overlay network [15, 22, 70]. To support historic subscriptions, databases are attached to a subset of brokers as shown in Figure 6.1. Each database is managed by a database administrator. Administration consists of assigning publication space partitions to one or more databases. A partition may be assigned to multiple databases to achieve replication, and multiple partitions may be assigned to the same database if database consolidation is desired. Partition assignments can be modified at any time, and any replicas will synchronize among themselves. The only constraint is that each partition be assigned to at least one database so no publications are lost.

We first outline how partitions are assigned to databases, and then explain the consistency model of database replicas followed by techniques used to maintain synchronization.
of replicas according to the consistency model.

**Partition Assignment:** Each database subscribes to `DB_CONTROL` publications addressed to it, and the administrator assigns partitions to databases by sending publications with `STORE` commands to the appropriate database. For example, the following publication assigns a partition to database `DB_A`.

```sql
INSERT (class, command, db, partition_spec)
VALUES (DB_CONTROL, STORE, DB_A, 'SELECT * WHERE shipID = *, level< 10')
```

The partition specification is itself a subscription with the selection formula expressed in the `WHERE` clause. A database that receives the `STORE` command will extract the partition specification and issue it as an ordinary future subscription. Matching publications will then be delivered to the database which will store them. A database assigned a partition also issues an advertisement that defines its partition.

When the first broker receives a historic subscription issued by a subscriber, it assigns it a unique query identifier and then routes the subscription as usual towards publishers whose advertisements intersect the subscription. This ensures that the subscription will arrive at databases whose partitions intersect the subscription. The database(s) convert the subscription into a SQL query, retrieve matching publications from the database, and publish the results. These “historic” publications are annotated with the subscription’s unique query identifier so they are only delivered to the requesting subscriber. After the result set has been published, the database will issue an `END` publication, which is used to unsubscribe the historic subscription.

The interaction with the databases fully leverages the content-based publish/subscribe model, and the databases are never addressed directly. In fact, it is impossible for publishers to discover where their publications are being stored, or for subscribers to know which databases process their queries. This simplifies management since databases can be moved, added or removed, and partitions reassigned at will.

**Partition Replication:** To improve availability, fault-tolerance and query perfor-
Chapter 6. Historical Data Access

mance, a partition may be replicated. Partition assignment strategies include partitioning, partial replication and full replication.

With partitioning, a database may be assigned several partitions, but each partition is assigned to only one database. That is, there is only one replica per partition. With partial replication, a given partition may be replicated by assigning it to multiple databases. The choice and location of replicas is studied in Section 6.3. With full replication, every database maintains replicas of all partitions. That is, each database stores all publications.

The various strategies have tradeoffs and are appropriate under different circumstances. The partitioning policy is simple and avoids replica consistency issues, but is sensitive to failures. Partial replication can tolerate failures of all but one replica, but requires logic to ensure the historic subscription is answered by only one of the replicas. Full replication is even more robust, and historic subscriptions can always be answered fully by the nearest database, minimizing network traffic. However, the high degree of replication imposes greater overall traffic and storage costs, as well as larger synchronization overhead. The partition assignment policies allow an administrator to tradeoff storage space, routing complexity, query delay, network traffic, parallelism of queries, and robustness.

**Consistency Model:** In standard publish/subscribe systems, two subscribers with identical subscriptions may receive notifications in different order. While the PADRES system will preserve the order of publications from any particular publisher, it cannot guarantee that subscribers observe the same interleaving of publications from multiple sources. Due to network delays, subscribers will typically receive publications from nearby sources sooner than those from distant ones. This is a generally accepted semantic in distributed publish/subscribe systems and many applications work with this assumption. Imposing a global ordering of publications will require centralized or quorum-based techniques that diminish the scalable distributed architecture.
In this thesis, we continue to provide this traditional publish/subscribe ordering semantic for both future and historic subscriptions. To clarify the discussion, we define various levels of consistency for subscription results.

**Definition** *Eventual Consistency*

There exist time periods $t_1$ and $t_2$ such that if two subscribers issue identical subscriptions at least $t_1$ time before any publisher, then after a period of $t_2$ after all publishers have stopped publishing, the publications delivered to the two subscribers are identical.

Notice that the eventual consistency definition does not specify anything about the order in which publications are delivered. As stated earlier, it is possible in distributed publish/subscribe systems to ensure per-source ordering of publications to interested subscribers. Let $p' \prec p$ mean that the same publisher issued publications $p'$ and $p$ and that it published $p'$ before $p$. Then, all subscribers that receive both $p'$ and $p$ will receive $p'$ first. However, subscribers may see a different order of interleaved publications from multiple sources. We define weak consistency to describe this generally accepted publish/subscribe semantic.

**Definition** *Weak Consistency*

Consider any pair of subscribers $s_1$ and $s_2$ that have issued identical subscriptions (at any time) and have received matching publication sets $P_1$ and $P_2$, respectively. For any pair of publications $p'$ and $p$ from the same publisher that appear in both $P_1$ and $P_2$, where $p' \prec p$, both subscribers received $p'$ before $p$. In addition, if there exists a $p''$ that matches both $s_1$ and $s_2$ and satisfies $p' \prec p'' \prec p$, then $p''$ must belong to both $P_1$ and $P_2$.

Informally, weak consistency states that all interested subscribers receive publications from a single source in the order the publications were published, and there are no “gaps” in the per-source publication stream. Note that the subscribers may receive different sets of publications because they may issue their subscriptions at different times.
Since subscribers do not know when data sources have stopped publishing, in practice, they can only rely on weak consistency semantics as opposed to eventual consistency.

In our system, the databases act as ordinary subscribers so database replicas can rely on, and hence provide, weak consistency semantics for historic subscriptions. Note that the publications sinksed into a database replica are inserted in an append-only manner, and there is only one “writer” per advertisement, so we avoid write conflicts even when a database receives publications from multiple publishers.

**Partition Replica Management**: Our system allows an administrator to add and remove replicas of a partition at any time, as well as **PAUSE** and **RESUME** a database. Consequently, database replicas may issue their subscriptions and sink publications during different periods of time. However, we require that a replica be able to deliver results for historic queries for its assigned partition, and the results must include all publications ever published since the system became operational (modulo the weak consistency allowance among replicas), including those publications issued before the replica was provisioned. To capture this point, we define a stronger consistency requirement for database replicas.

**Definition Replica Consistency**

The publications stored in any pair of replicas $r_1$ and $r_2$ of a partition $R$ must (i) be weakly consistent and (ii) if a publication $p$ appears in both $r_1$ and $r_2$, then for every publication $p'$ from the same publisher that belongs to the partition and $p' \prec p$, $p'$ must appear in both $r_1$ and $r_2$.

Informally, replica consistency adds the requirement that replicas contain a complete history of publications before some point in time.

Since partition replicas may be added and removed at any time, they need to synchronize among themselves to maintain the replica consistency. We assume that every partition is always assigned to at least one replica so that every publication is available in at least one database.
Databases transition among three states for each of the partitions assigned to them: **synchronizing**, **active**, and **paused**. A database can only answer historic subscriptions for a partition when it reaches the **active** state. When a new partition is assigned to a database, it begins in the **synchronizing** state and attempts to transition to the **active** state. A transition from **synchronizing** or **paused** states to the **active** state invokes a synchronization protocol so that the partition maintains replica consistency with its replicas.

Synchronization consists of four steps. First, the database issues the partition specification as a subscription as described earlier. *Ack* messages are collected from the leaves of the subscription propagation tree and are hierarchically propagated back to the database. The acknowledgments also include a count of potential data sources $N_S$ for the partition. The receipt of the cumulative acknowledgment confirms that the subscription has been propagated throughout the overlay and any future publications will (eventually) be delivered to the database. Any publications $P_{new}$ received by the database before the protocol is complete is buffered. In addition, upon receiving a partition specification subscription, a broker with a publisher $p$ connected to it that has submitted an advertisement that intersects the subscription now issues a **FLUSH** publication to all database replicas whose partitions intersect the advertisement. The delivery of this publication at an active database replica confirms that all publications issued by publisher $p$ before **FLUSH** have been delivered to the database. Third, the new database replica requests all the publications $P_{old}$ stored at an active replica. The existing active replica will respond to the request only after receiving $N_S$ corresponding **FLUSH** publications. (The value of $N_S$ was included as part of the request from the new replica.) Finally, the new replica will interleave publications $P_{old}$ and $P_{new}$ so they respect weak consistency ordering (i.e., per-source ordering), filter out duplicate publications, and insert them into the database.

4As an optimization, a paused replica may send, along with the request, the most recent publication $p_s$ it has from each source $s$ to indicate that it only needs publications $p_s'$ published after $p_s$: $p_s \prec p_s'$. The assumption is that the replica was replica consistent before it was paused.
Publishers tag their publications with an increasing sequence number inserted to facilitate this ordering. At this point the newly provisioned (or resumed) partition is replica consistent with any other active replica, and it transfers to the active state and begins to answer historic queries. The correctness of this protocol is established by the property below.

**Property 1.** The synchronization protocol guarantees replica consistency.

*Proof.* Consider every publication \( p \) that appears in both \( r \) and \( r_a \) immediately after the synchronization protocol completes. There are two cases:

Case 1: \( p \in P_{\text{old}} \). Since \( r_a \) is active and replica consistent, all \( p' \prec p \) will be in \( P_{\text{old}} \). Since \( r \) includes all publications from \( P_{\text{old}} \), \( p' \) is also in \( r \).

Case 2: \( p \not\in P_{\text{old}} \). This means that \( p \) was published by some source \( S \) after the FLUSH publication \( p_S^f \) from \( S \): \( p_S^f \prec p \). Since \( p_S^f \) is only sent after the partition specification subscription from \( r \) has propagated to \( S \), publish/subscribe routing will guarantee that \( p \) is delivered to \( r \). In other words, \( p \in P_{\text{new}} \). Now, for every \( p' \prec p \), either \( p' \prec p_S^f \) or \( p_S^f \prec p' \). In the former case, \( p' \in P_{\text{old}} \) and both replicas contain \( p' \). In the latter case \( p' \in P_{\text{new}} \) (for the same reason that \( p \in P_{\text{new}} \)), then \( p' \) is in \( r \). On the other hand, because \( p' \prec p \) and \( p \) is in \( r_a \), and we assume per-source ordered delivery, then \( p' \) is also at \( r_a \).

Since in all cases, both replicas contain all publications \( p' \prec p \), and the replicas can order the publications per source (from the sequence numbers annotated to publications), the protocol guarantees replica consistency.

\[\square\]

### 6.2.3 Dynamic Partitions

As the subscription and publication workload changes, it may be worthwhile to repartition the publication space. Once a new partitioning scheme has been computed, the administration must assign these new partitions to either existing or new databases. The
databases will synchronize among themselves as outlined above, so that the new partitions retrieve their relevant data from one or more old existing partitions. Once all the new partitions are active, the administration may shut down the old partitions.

![Partition Overlaps](image)

**Figure 6.3: Partition Overlaps**

During this transition period, there may be times where both old and new partitions are running in the system. In particular there may be partitions with overlapping advertisements. Consider Figure 6.3 where a subscription $S$ intersects two overlapping partitions with advertisements $A_1$ and $A_2$. Assume $I = P(S) \cap P(A_1) \cap P(A_2)$ is not empty. Now, publications not in $I$, must be retrieved from the appropriate database, but those in $I$ may be retrieved from either. Anytime a subscription retrieves data from a partition with advertisement $A_i$ that overlaps some other partitions, the database is given, in addition to $S$, an exclusion set of advertisements $\bar{A}_i$. Publications that belong to advertisements in this exclusion set are discarded from the result set returned by the database. Formally, if there exists an advertisement $A \in \bar{A}_i$ such that publication $p \in P(A_i)$ and $p \in P(A)$, then $p$ is discarded. The next section describes how the system decides which publications to retrieve from which database partitions.

### 6.2.4 Partition Selection Cost Model

When partitions are replicated or overlapping, the database from which to retrieve publications affects the subscription evaluation performance, including network traffic and database load distribution. This section presents functions that quantify the cost of re-
CHAP.

Chapter 6. Historical Data Access

109

trieving data from various partitions, and heuristics to minimize these cost functions. Note that Section 7.1.3 models the cost of composite subscription correlation location, whereas this section quantifies the cost of a decomposition of an atomic subscription among partitions.

Given a set of partitions represented by advertisements \( \{A_1, \ldots, A_n\} \) that intersect a subscription, the problem is to decide what subset of \( P(A_i) \) should be retrieved from partition \( i \). Formally, we want a transformation function \( F \) that maps each \( A_i \) to an \( A'_i \) such that \( \bigcup_i P(A_i) = \bigcup_i P(A'_i) \) and \( P(A'_i) \cap P(A'_j) = \emptyset \) for all \( i \neq j \). More specifically, the \( A'_i \) are computed using an exclusion set \( \bar{A}_i : A_i \xrightarrow{F} A'_i = A_i - \bar{A}_i \).

We seek a function \( F \) that minimizes some metric. For example minimizing the network cost of retrieving data from the databases can be formalized as

\[
\min_F \sum_i |P(S) \cap P(A'_i)| d(A_i),
\]

where \( d(A_i) \) is the network cost of retrieving a publication from database partition \( A_i \).

For this optimization metric, a greedy algorithm that always selects the nearest database partition results in an optimal selection of partitions. Evaluations in Section 6.3 demonstrate the significant savings this algorithm provides when partition replicas exist.

Another optimization is to minimize the number of databases queried. This also has the effect of reducing the exclusion set size \( |\bar{A}_i| \). Large exclusion sets increase the complexity of processing the query at the database, since either the query must be rewritten so as not to retrieve publications in the exclusion set, or more joins need to be computed. This optimization can be expressed as

\[
\min_F \sum_i NE(P(S) \cap P(A'_i)),
\]

where function \( NE(X) \) (not empty) evaluates to 1 if set \( X \neq \emptyset \), and 0 otherwise. A greedy algorithm like the one used earlier would not optimize this metric. Instead, we use a heuristic in which the partition advertisements are sorted by the cardinality of the advertisement space that intersects the subscription: \( |P(S) \cap P(A_i)| \). Then the exclusion set for each advertisement are simply those with lower rank: \( A'_i = \{A_1, \ldots, A_{i-1}\} \).
Intuitively, this heuristic prioritizes partitions with large result sets.

To fully answer a historic subscription, data from every database partition queried must be retrieved and returned to the subscriber. Therefore, another strategy is to minimize the time needed to retrieve and transfer publications from the slowest database (i.e., the database with the largest result set). Formally,

$$\min_F (\max_i |P(S) \cap P(A'_i)|).$$

This optimization function is motivated by the results in Section 6.3 which show that significant impact on notification delay when a large part of a historic subscription is answered by a small number of databases. A heuristic to approximate an optimal solution to this metric works in a manner similar to the previous one where the advertisements are first ranked by their cardinality. However, this time we favour advertisements with smaller cardinalities. Therefore, the exclusion sets for $A'_i$ are $\{A_{i+1}, \ldots, A_n\}$.

Some metrics may conflict one another. For example, favouring the nearest database or attempting to minimize the slowest database may cause more databases to be queried. To consider these tradeoffs, a cost metric that minimizes the total subscription processing time considering all three factors above can be expressed as

$$\min_F (\max_i [C|P(S) \cap P(A_i)||\bar{A}| + |P(S) \cap P(A'_i)|d(A_i)])$$

Here $C$ represents a constant for the average time to retrieve a historic publication from the database. The function also makes a worst case assumption on the exclusion set processing, namely that publications retrieved from the database are individually filtered by each advertisement in the exclusion set. Our approximate solution here is similar to the previous one. This time, we first rank the partition advertisements not simply based on the expected cardinality, but by the estimated time to query the database and transfer the publications to the subscriber. The estimated query time is the expression inside the max function in the above optimization function. Then, we compute the exclusion sets for each advertisement as in the previous metric.
6.3 Evaluation

This section evaluates an implementation of the subscription routing (Section 6.1) and partitioning (Section 6.2) algorithms in Padres. A 30 broker publish/subscribe overlay (and one database attached to each broker) with 10 publishers and 20 subscribers are deployed across a 20 node cluster of 1.86 GHz machines with 4 GB of RAM. This size of topology is representative of the border security scenario described earlier. Also, the publications used are derived from this scenario, and subscriptions generated with predicates following a Zipf distribution model locality among subscribers.

We investigate the following policies: centralized (all partitions in one database replica), partitioning (one replica per partition per database), partial replication (each database is still assigned at most one partition but 30%, 60% or 100% of the partitions have two replicas), full replication (every partition assigned to every database) and merged partial replication (60% of partitions have two replicas, and two partitions are assigned per database). A centralized policy in which a single database replica manages all partitions serves as a baseline. In all cases, the global publication space is divided into five partitions.

Publication storage: We first evaluate the performance of mapping between publish/subscribe messages and SQL statements. To store a publication in a database, the publication is converted into an SQL \texttt{INSERT} statement. The main factor here is the number of attributes in the publication. Table 6.1 shows that the storage time (averaged over 5000 publications for each data point) increases linearly with the number of predicates.

<table>
<thead>
<tr>
<th>predicates / publication</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ms)</td>
<td>121</td>
<td>136</td>
<td>217</td>
<td>239</td>
<td>280</td>
</tr>
</tbody>
</table>

Subscription processing: A historic subscription arriving at a database is converted into an SQL \texttt{SELECT} statement, and each query result is transformed into a pub-
lication message. Figure 6.4 quantifies this time but excluding sending the publications. Results show that this processing time increases roughly linearly with the number of subscription predicates and the result set size.

![Query Processing Time](image)

**Figure 6.4: Query Processing Time**

**Storage and Query Traffic:** Publications must be routed to interested subscribers and to all corresponding database replicas. The latter cost depends on the path length between the publisher and databases, and increases (sublinearly due to publish/subscribe multicasting) with the number of replicas. Figure 6.5 shows that the number of overlay hops traversed by publications increases with the number of replicas. The partitioning scheme has no replicas and each publication is routed to exactly one database, whereas publications are broadcast to all databases under full replication.

Whereas a publication must be routed to all replicas of a single partition, a historic query is processed by a single replica of all relevant partitions. This affords opportunities for parallelism (if partitions are assigned to different databases) and path optimization (by choosing a closer replica). We use atomic historic subscriptions with two to five predicates each, each of which access between two and four partitions. In contrast with
Figure 6.5: Partition Sinking Traffic

Figure 6.6: Notification Delay
the storage traffic, Figure 6.5 shows that total query result publication traffic decreases with more replicas. Compared to the partitioning strategy, partial replication of 30%, 60%, and 100% saves 7.9%, 36.1%, and 59.1% of the query traffic, respectively. Full replication performs best since all publications can be retrieved from the database at the subscriber’s host broker. In the centralized case, all storage and query traffic go to a single database, and neither metric can be improved by a different selection of replicas. The first optimization metric from Section 6.2.4—which optimizes the subscription network traffic—was used in the above experiments, and the results indicate that nearby replicas are indeed used and decrease the query traffic.

There is a clear tradeoff between storage and query traffic when varying the number of replicas in Figure 6.5. If network traffic is the primary concern, the choice of the replication policy will depend on the bias of the workload towards writes (publications) or reads (subscriptions).

**Notification Delay:** The replication policy not only affects query traffic, but also the time to deliver query results to subscribers. Figure 6.6 shows the average delay of processing subscriptions with increasing result sets. The delay is computed as the period from when a subscriber issues a subscription to when it receives a matching notification, averaged over all notifications. Each subscription accesses an average of four partitions. It may be surprising that the full replication policy suffers one of the longer delays despite all the data being available at the closest database. We mentioned earlier that queries with larger result sets take longer to process. With full replication, the entire query is answered by a single database whereas the other policies process queries with smaller result sets in parallel at multiple databases. The centralized policy is even worse than full replication since it not only suffers from a large query result set but provides no choice in selecting a closer replica. Policies with fewer partitions per database perform better, indicating that the query processing time at the database dominates the time to route the results to the subscriber. Among the policies that assign one partition per
database (partitioning and partial replication), Figure 6.6 shows that increasing replicas decreases the delay. In this case, the query processing delay remains constant but more replicas provide opportunities to decrease the routing delays by selecting a nearby replica. However, the benefits of replication are small relative to those of parallelism, reinforcing the observation that query processing delays dominate routing delays.

Figure 6.7 presents the delay results from Figure 6.6 in more detail, now showing the period of time between the first and last notification at the subscriber. We see that even though full replication routes results faster than with partitioning (the time range between the first and last notification is smaller), the query processing time is worse (the first notification takes longer to deliver). For example, in the case of subscriptions with about 18,000 matches, full replication requires about 30 s to deliver the first notification compared to less than 0.5 s with partitioning. We also see that the first notification is delivered just as fast with partial replication as with partitioning since they both exploit parallelism, but the former delivers the final notification sooner since it benefits from database locality. These results support the assumption underlying the third metric in Section 6.2.4 which tries to minimize the number of results returned by the database that returns the most results, effectively balancing the query load across replicas and achieving greater parallelism.

The observations confirm that database processing delays and parallelism benefit more than locality and routing path lengths. They also suggest that if notification delay as opposed to network traffic is to be optimized, queries should not simply be processed by the nearest replica, but query result set size and parallelism should be considered.

**Partitioning:** Algorithm 2 produces partitions that are evenly accessed. Comparing this even partitioning with one which randomly divides the publication space, Figure 6.8 shows that the average notification delay with even partitioning is 61% better. Also, since the number of publications from each partition under uneven partitioning vary widely, the lightly loaded partitions delivery notifications soon, but heavily loaded partitions are
Chapter 6. Historical Data Access

Figure 6.7: Delay Range

Figure 6.8: Partitioning
slow and ultimately cause the average delay to be worse. In the centralized approach and the full replication, where all publications are stored in one database, the processing time of a query dominates the notification delay. Full replication performs 12% better than the centralized approach because it decreases the routing delay by using the local database.

**Hybrid Subscriptions:** In Figure 6.9 we examine the notification delay for a hybrid atomic subscription that queries the databases (the past) and two publishers (the future). The figure plots the average delay of the $n^{th}$ notification. The relative delays of the historic notifications in the presence of active publishers conforms to the earlier results: delay increases with more partitions per database. It is interesting to observe the hills and valleys for some of the schemes in Figure 6.9, such as the partitioning policy, where we can see bursts of notifications arriving from different databases.

We also evaluate a hybrid composite subscription that correlates historic and future publications. Without this capability, a subscriber must subscribe to the future publications, and for each publication it receives generate an SQL statement to query the database for correlated data. We refer to this as the baseline approach. With hybrid composite subscriptions, however, the correlation is done by the brokers in the network, and the subscriber is only notified of the correlation results. Figure 6.10 compares the network traffic for both approaches, where $N$ is the number of (historic) publications in the database and $M$ is the number of (future) publications from publishers. The x-axis is the average selectivity of a publication over the historic data, and controls the correlation between historic and future data. The results show that when selectivity is high, that is, there is little correlation between historic and future publications, the baseline approach incurs less traffic since the uncorrelated historic publications are not forwarded. The drawback with the baseline approach is that the subscriber must receive all future publications (even those not correlated with any historic publications), and issue SQL queries for each. With hybrid composite subscriptions, this work is done by the brokers.
Figure 6.9: Hybrid Atomic Subscription

Figure 6.10: Hybrid Composite Subscription
in the network. When both $N$ and $M$ are large, and the correlation is high, Figure 6.10 shows that the in-network approach saves up to 20% of the network traffic.
Chapter 7

Cost Model

In the extended publish/subscribe model, publication traffic includes future data from publishers and historic data from databases. Usually, publications from databases are heavier than that from publishers. As a result, a routing algorithm which takes the network traffic into account is necessary in the new model, especially for composite subscriptions. For example, if a composite subscription correlates two publication data sources, one is from a publisher and the other is from a database. The number of publications from the database outweighs that of the publisher. When such unbalanced traffic is involved, the composite event detection is less costly if the detection is moved close to the heavy publisher, e.g., the database in this case. A cost model is needed to estimate the cardinality of matching publications and the detection cost.

The cost model is used to make optimal solutions based on the knowledge we have. In content-based publish/subscribe, components are loosely coupled. That is, publishers do not know where and how many subscribers are, and vice versa. As a result, no global knowledge is available in the system. Every broker or clients only know their neighbors. And since content-based matching operation is expensive, global knowledge is not feasible. In this section, we propose the cost model based on local information which is easier to achieve and cheaper to compute. The routing decisions made based on
the cost model is local optimal. The algorithms we propose are greedy algorithms. We assume that these greedy algorithms can reach “near” global optimal solutions.

## 7.1 Cost Model

### 7.1.1 Model Description

A broker routing a composite subscription makes local optimal decisions based on the knowledge available at itself and its neighbors. The cost function can capture the use of resources such as memory, CPU, and communication. Suppose composite subscription $CS$ is split at broker $B$. The total routing cost of $CS$ is

$$TRC_B(CS) = RC_B(CS) + \sum_{i=1}^{n} RC_{B_{N_i}}(CS_{B_{N_i}}),$$

and includes the routing cost of $CS$ at broker $B$ and those neighbors where publications contributing to $CS$ may come from. $CS_{B_{N_i}}$ denotes the part of $CS$ routed to broker $B_{N_i}$, and may be an atomic or composite subscription.

**Routing cost at each broker:** The cost of a composite subscription $CS$ at a broker includes not only the time needed to match publications (from $n$ neighbors) against $CS$, but also the time these publications spend in the input queue of the broker, and the time that matching results (to $m$ neighbors) spend in the output queues. This cost is modeled as

$$RC_B(CS) = \sum_{i=1}^{n} T_{in} |P(CS_{B_{N_i}})| + \sum_{i=1}^{n} T_{m} |P(CS_{B_{N_i}})| + \sum_{i=1}^{m} T_{out_i} |P(CS)|,$$

where $T_m$ is the average matching time at a broker, $T_{in}$ and $T_{out_i}$ are the average time messages spend in the input queue, and output queue to the $i^{th}$ neighbor. $|P(S)|$ is the cardinality of (atomic or composite) subscription $S$, and is defined below. To compute the cost at a neighbor, brokers periodically exchange information such as $T_{in}$ and $T_m$. This information is incorporated into an M/M/1 queuing model to estimate queueing times at neighbor brokers as a result of the additional traffic attracted by splitting a composite subscription there.
Let \( D = \int_{\text{Max}(a_i)}^{\text{Min}(a_i)} d_{a_i} \)

\[
sf_A(a_i = \text{val}) = \frac{d_{a_i}(\text{val})}{D} \\
sf_A(a_i > \text{val}) = \int_{\text{val}}^{\text{Max}(a_i)} d_{a_i}/D \\
sf_A(a_i < \text{val}) = \int_{\text{Min}(a_i)}^{\text{val}} d_{a_i}/D
\]

Table 7.1: Selection Factor

\[
|P(CS)| = \begin{cases} 
|P(S_l)| + |P(S_r)| & \text{if } op = \|; \\
\min(|P(S_l)|, |P(S_r)|) & \text{if } op = &.
\end{cases}
\]

Table 7.2: Subscription Cardinality

**Subscription cardinality:** The cardinality a subscription \( S \) is the number of matches of \( S \) per unit time, that is, the frequency of match. But first, we define the *selection factor* of subscription \( S \). If \( S \) is atomic, its selection factor with respect to advertisement \( A \) is defined as

\[
sf_A(S) = \prod_{i=1}^k sf_A(p_i), \text{ where } p_i \text{ is the predicate of attribute } a_i \text{ in } S \text{ and } sf_A(p_i) \text{ is the selection factor of predicate } p_i.
\]

The selection factors of individual predicates are computed based on the predicate operator, and the distribution of attribute values \( d_{a_i} \) across publications, as shown in Table 7.1.

An advertisement’s attribute distributions are disseminated as a histogram as part of the advertisement. We note two sources of inaccuracy in the selection factor estimation that arise from having to tradeoff accuracy with cost. First, the above equations do not consider the joint distribution among attributes, which would improve the estimation but require more information to be disseminated. Also, the accuracy of the attribute distributions themselves depends on the histogram bucket size and frequency of updates. We leave the exploration of these tradeoffs for future work.

We can now calculate the cardinality of an atomic subscription \( S \) that intersects advertisements \( \{A_1, A_2, \ldots, A_q\} \), as \(|P(S)| = \sum_{i=1}^q r_i \ast sf_{A_i}(S)\), where \( r_i \) is the publication rate associated with advertisement \( A_i \). The cardinality of a composite subscription \( CS = S_l \ op \ S_r \) is shown in Table 7.2.
7.1.2 Example

We now give an example of how the DCSR algorithm can route composite subscriptions based on the publication traffic and the status of the overlay. Fig. 7.1 shows a possible routing solution for composite subscription $CS = S_1 \& S_2$, where the $S_i$ are atomic. At Broker 6, $CS$ is routed as a whole towards Broker 4, where the destinations of both $S_1$ and $S_2$ are different, and as a result, Broker 4 is the join point broker of $CS$ in the topology-based routing algorithm. If the amount of data from $S_1$ is significantly larger than that from $S_2$, it may be more efficient to evaluate $CS$ at Broker 3 rather than Broker 4. In the DCSR algorithm this dynamic evaluation occurs at each broker until a broker decides to split the composite subscription, say at Broker 1, in which case Broker 1 becomes the join point broker for $CS$.

Since network conditions, such as delay and bandwidth, may change, the join point chosen by the DCSR algorithm may not remain optimal, and should be computed dynamically. If the join point broker finds a broker that is able to detect the composite event with a cost lower than itself, it initiates a join point movement [55].

The DCSR algorithm determines the location that minimizes the network traffic and message delay costs of evaluating composite subscriptions. The DPR algorithm from Section 5.3.3 further reduces the cost of composite event detection.

**Adaptive Routing:** Topology-based composite subscription routing [52] evaluates correlation constraints in the network where the paths from the publishers to subscribers merge. If a composite subscription correlates a historic data source and a publisher, where the former produces more publications, correlation detection would save network traffic if moved closer to the database, thereby filtering potentially unnecessary historic
publications earlier in the network. Based on this observation, we propose the adaptive composite subscription routing algorithm described next.

The **WHERE** clause constraints of a composite subscription can be represented as a tree where the internal nodes are operators, leaf nodes are atomic subscriptions, and the root node represents the composite subscription. The composite subscription example is represented as a tree in Figure 6.1. A recursive algorithm computes the destination of each node in the tree to determine how to split and route the subscription. The algorithm traverses the tree as follows: if the root of the tree is a leaf, that is, an atomic subscription, the atomic subscription’s next hop is assigned to the root. Otherwise, the algorithm processes the left and right children’s destination trees separately. If the two children have the same destination, the root node is assigned this destination, and the composite subscription is routed to the next hop as a whole. If the children have different destinations, the algorithm estimates the **total routing cost** (to be presented shortly) for potential candidate brokers, and the minimum cost destination is assigned to the root. If the root’s destination is the current broker, the composite subscription is split here, and the current broker is the **join point** and performs the composite event detection. The algorithm assigns destinations to the tree nodes bottom up.

**Dynamic Join Point Movement:** When network conditions change, join points may no longer be optimal and should be recomputed. A join point broker periodically evaluates the cost model, and upon finding a broker able to perform detection cheaper than itself, initiates a join point movement. The state transfer from the original join point to the new one includes routing path information and partial matching states. Each part of the composite subscription should be routed to the proper destinations so routing information is consistent. Publications that partially match composite subscriptions stored at the join point broker must be delivered to the new join point.
7.1.3 Join Point Placement Cost Model

A broker routing a composite subscription makes locally optimal decisions based on a cost model that incorporates its own resource consumption and that of its neighbors. Resources such as memory, CPU, and network traffic can be modeled. Suppose composite subscription $CS$ is split at broker $B$. The total routing cost for $CS$ is

$$TRC_B(CS) = RC_B(CS) + \sum_{i=1}^{n} RC_{B_{N_i}}(CS_{B_{N_i}}),$$

and includes the routing cost of $CS$ at broker $B$ and those neighbors with publications contributing to the composite subscription. $CS_{B_{N_i}}$ denotes the part of $CS$ routed to broker $B_{N_i}$, and may be atomic or composite.

**Routing Cost at Each Broker:** The cost of $CS$ at a broker includes the time to match publications (from $n$ neighbors) against $CS$, the time these publications spent in the broker’s input queue, and the time that matching results (to $m$ neighbors) spend in output queues:

$$RC_B(CS) = \sum_{i=1}^{n} T_{in} * |P(CS_{B_{N_i}})| + \sum_{i=1}^{n} T_{matching} * |P(CS_{B_{N_i}})| + \sum_{i=1}^{m} T_{out,i} * |P(CS)|,$$

where $T_{matching}$ denotes the average matching time at a broker, and $T_{in}$ and $T_{out,i}$ are the average time messages spent in the input queue and the output queue to the $i^{th}$ neighbor. $|P(S)|$ is the cardinality of (atomic or composite) subscription $S$, and is defined below.

To compute the cost at a neighbor, brokers periodically exchange $T_{in}$ and $T_{matching}$, and use this information in an M/M/1 queuing model to estimate queuing times at neighbor brokers as a result of the additional traffic attracted by splitting a composite subscription there.

**Subscription Cardinality:** Different from the query cardinality in database area which is evaluated against existing records, we define the cardinality of a subscription $S$ is the number of matches of $S$ per unit of time. Let the attribute set of advertisement $A$ be $Attr(A) \Leftarrow \{a_1, a_2, ..., a_n\}$. To evaluate the cost model we need the cardinality of
\( \wp(A) \) per attribute \((|\Pi_{a_i}\wp(A)|)^{1}\) and the distribution of values in an attribute domain \((\text{dom}[a_i])\).

To estimate the cardinality of a publication set matching a subscription \(s\), we define a selection factor as \(|\sigma_s \wp(A)| = sf_A(s) \ast |\wp(A)|\). An atomic subscription’s selecting factor is \(sf_A(s) = \prod_{i=1}^{n} sf_{A_i}(p_i)\), where \(p_i\) is the predicate of attribute \(a_i\) and \(sf_{A_i}(p_i)\) is the selecting factor of predicate \(p_i\) in subscription \(s\). The selection factors of individual predicates are computed based on the predicate’s operator and the distribution of attribute values \(d_{a_i}\) across publications, as shown below.

Let \(D = \int_{\text{Min}(\text{dom}[a_i])}^{\text{Max}(\text{dom}[a_i])} d_{a_i}\)

\(sf_A(a_i = val) = d_{a_i}(val)/D\)

\(sf_A(a_i > val) = \int_{\text{val}}^{\text{Max}(\text{dom}[a_i])} d_{a_i}/D\)

\(sf_A(a_i < val) = \int_{\text{Min}(\text{dom}[a_i])}^{\text{val}} d_{a_i}/D\)

An advertisement’s attribute distributions are disseminated as a histogram as part of the advertisement.

We can now calculate the cardinality of an atomic subscription \(S\) that intersects advertisements \(\{A_1, A_2, \ldots, A_q\}\), as \(|P(S)| = \sum_{i=1}^{q} r_i \ast sf_{A_i}(S)\), where \(r_i\) is the publication rate associated with advertisement \(A_i\).

The selection factor of a composite subscription with an attribute join is \(sf(A_1 \bowtie_{a_i} A_2) = \frac{|\Pi_{a_i}A_1 \bowtie_{a_i} A_2|}{|\wp(A_1)| \ast |\wp(A_2)|}\). The cardinality of a composite subscription \(CS = S_l \ op \ S_r\) is:

\[
|P(CS)| = \begin{cases} 
|P(S_l)| + |P(S_r)| & \text{if } op = \parallel; \\
\min(|P(S_l)|, |P(S_r)|) & \text{if } op = \&; \\
sf(A_1 \bowtie_{a_i} A_2) \ast |s_1 \bowtie_s s_2| & \text{if } op = \bowtie_{a_i}.
\end{cases}
\]

\(^{1}\text{Estimated based on publications over a period of time.}\)
7.2 Evaluation for General Overlays

Dynamic CS Routing  We evaluate the DCSR algorithm on PlanetLab using a topology similar to that in Fig. 7.1. Twenty publishers publish at rates ranging from 100 to 600 per minute, and 30 subscribers issue subscriptions, with one of them being a composite subscription. In Fig. 7.2 we measure the bandwidth of certain brokers located on the composite subscription routing path. The solid bars represent the number of outgoing messages at a broker, and the hatched bars are the number of incoming messages that are not forwarded. Note that the sum of the solid and hatched bars represents the total number of incoming messages as a broker. We also measure notification delays in Fig. 7.3, as measured from when the last publication contributing to the composite subscription is published to when the subscriber receives the notification.

The composite subscription is issued at Broker 6 in Fig. 7.1, and is routed to its potential publishers using simple routing, topology-based routing or DCSR. In simple routing, a composite subscription is split into atomic subscriptions at the broker that first receives the composite subscription from a subscriber. In this case, the split occurs at Broker 6, and all publications must be routed to this broker where the composite subscription is evaluated and unmatched publications finally filtered out. This is illustrated in Fig. 7.2 where we see that with simple routing, only Broker 6 filters any messages, and
therefore all preceding brokers incur higher than necessary message load. In topology-based routing, however, Broker 4 is the join point broker, and we observe the filtering that occurs here in Fig. 7.2, as well as the reduced message loads at Brokers 5 and 6. The DCSR algorithm determines the composite subscription detection location based on the potential publication traffic. In our topology, the publisher at Broker 1 has a higher publication rate, and hence this broker is an efficient point to detect the composite subscription. Fig. 7.2 shows that filtering indeed occurs at Broker 1 and that all subsequent brokers enjoy a reduced message load.

To summarize, the topology-based composite subscription routing algorithm imposes less traffic than simple routing by moving the join point into the network, and the DCSR algorithm further reduces traffic by moving the join point closer towards congested publishers as indicated by the cost model. In the scenario in Fig. 7.2, compared to simple routing, the DCSR algorithm reduces the traffic at Brokers B1 by 79.5%, a reduction that is also enjoyed by all brokers downstream of the join point.

In Fig. 7.3 we see that with the DPR algorithm, the simple composite subscription routing performs the worst, and the DCSR the best. Even compared to the topology-based approach, the DCSR algorithm manages to reduce the notification delay by 55%, by filtering out messages early in the network and hence reducing queueing delays. In this scenario, we also evaluate fixed and dynamic publication routing with the DCSR algorithm. Fig. 7.3 shows that the dynamic approach improves the delay by 40.1% compared to fixed publication routing. We expect the benefits to be even more pronounced in larger networks since longer composite subscription paths in such topologies offer the potential for more savings in terms of traffic and delay.
7.3 Evaluation for Historic Data Access

We evaluate the composite subscription routing protocols from Section 6.1: simple, topology-based, and adaptive routing. A composite subscription is issued that correlates data from eight sources: seven publishers publishing at rates from 50 to 500 msg/min, and one database. We measure the network traffic and notification delay as measured from the publication time of the last publication that contributes to the composite subscription match to the time when the subscriber is notified of the match. In both Figures 7.4 and 7.5, adjacent brokers (those one unit away in the row and/or column axes) are neighbors in the overlay. Also, the composite subscriber is connected to the broker at (column,row) coordinate (10,8) and publishers are connected to the brokers at (*,1).

Figure 7.4 plots the outgoing traffic at each broker in the overlay. In simple routing, a composite subscription is split into atomic subscriptions at the broker that first receives the composite subscription from a subscriber. In this case, broker (10,8) evaluates the composite subscription and filters out unmatched publications. Since filtering does not occur until the last broker along the paths from publishers to subscriber, each broker along the path from publishers to the subscriber has high outgoing traffic.

In topology-based routing, a composite subscription is forwarded as a whole until it reaches the broker where its atomic subscriptions must be forwarded in different directions in the topology. In our topology, the composite subscription is split at brokers (1,5) and (12,5), and we observe in Figure 7.4 that filtering occurs at those brokers.

The adaptive routing algorithm determines the composite detection location based on potential publication traffic. In this scenario, publishers with high publication rates connect to brokers (1,1) and (5,1), and hence it is desirable to detect composite subscriptions near them. The results in Fig 7.4 show that the adaptive algorithm reduces traffic by an average of about 66% and 43% compared to simple and topology-based routing, respectively. These savings are enjoyed by all brokers downstream of the join points. The
average notification delay in topology-based routing is about 0.1 s, which the adaptive algorithm manages to reduce by about 48% by filtering out messages early in the network and hence reduce queuing and routing delays.

**Dynamic workload:** We further evaluate adaptive routing in a scenario where conditions change. Publication rates are modified during the experiment: heavy publishers reduce their rates from 400 to 50 msg/min, and others increase their rates from 100 to 500 msg/min. Under the new workload, the original join point brokers initially chosen by the adaptive approach may no longer be optimal. Figure 7.5 shows traffic with the changing workload when the join points remain at their optimal locations as determined during subscription routing (the adaptive case), and when they are allowed to react to the changing workload (the dynamic case). For comparison, the figure also replots from Figure 7.4 the adaptive results under a static workload. In all cases, we only measure the traffic after the workload has changed.

When the join points do not move (adaptive cases), a change in the publication workload increases the overall traffic by about 220% since the join points are no longer optimal. However, by moving the join points to brokers (3,1) and (9,1), where more publications are being generated, the dynamic algorithm reduces the total network traffic by about 72% compared to the case when the join points remain fixed despite changing
conditions. Again, the traffic reductions are enjoyed by all brokers downstream of the new join points.

![Figure 7.5: Dynamic Workload](image-url)
Chapter 8

Case Study: Supporting BPEL

This chapter presents the case study of a business process management system, which is implemented on top of PADRES. Business process management is a fundamental component of any data-centric operation, and any gains made in this area have a far-reaching impact and benefit for enterprises. The Business Process Execution Language (BPEL) standardizes the development of composite enterprise applications that make use of software components exposed as Web services. BPEL processes are currently executed by a centralized orchestration engine, in which issues such as scalability, platform heterogeneity, and division across administrative domains can be difficult to manage. We propose NIÑOS, a distributed agent-based orchestration engine, in which several light-weight agents execute a portion of the original business process and collaborate in order to execute the complete process. The complete set of standard BPEL activities are supported in NIÑOS. The business process management system for BPEL processes performs three key tasks: process deployment, process execution and process monitoring. Process deployment is to install activities on agents distributed in the system network. Process execution is simply the invocation and control of BPEL activities using remote methods such as RPC and Web Services. Process monitoring involves maintaining a set of status information for each executing or previously executed activity.
8.1 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.

![NIÑOS Distributed Business Process Execution Architecture](image)

8.1.1 NIÑOS System Architecture

The NIÑOS system architecture, as shown in Figure 8.1, consists of four components: the underlying PADRES broker network, activity agents, Web service agents, and a business process manager. As mentioned in Section 2.3, 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 8.1.2 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 8.1, 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 a partial 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 future research directions for this system.
The Padres and NIÑOS system architecture is conceptually summarized in Figure 8.2. 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.

8.1.2 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 2.1, 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 8.3, 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 8.3. 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 8.3 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 8.3), 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 8.1.2.

**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 8.4. 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 completes. The *pick* agent listens for this event with subscription Sub1 in Figure 8.4. Also, the *pick* agent issues a subscription for each onMessage it listens for (Sub2 in Figure 8.4), and when a matching event occurs, it issues a publication to trigger the appropriate activity (Pub1 in Figure 8.4).

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.
Figure 8.3: BPEL While Activity

Figure 8.4: BPEL Pick Activity
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 8.5. 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 8.5, and triggers the first activity in its compensation handler using Pub1 in Figure 8.5.

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, 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 8.5, 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 is evaluated to be true is taken. If all the cases fail, an optional otherwise branch is taken.

Figure 8.6 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
Chapter 8. Case Study: Supporting BPEL

Figure 8.5: BPEL Compensate Activity

Figure 8.6: BPEL Switch Activity
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 8.6), 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 8.6. The first activity in each branch subscribes to these trigger publications. For example, in Figure 8.6, activity2 subscribes to Sub2. Note that the case where none of the cases in a switch activity is 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 8.6, 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 8.6, 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 8.7. 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 8.7), triggers the execution of each flow branch (Pub1 in Figure 8.7), 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 8.7), and publishes its completion as usual (Pub2 in Figure 8.7). Both activity2 and activity5 belong to this case in Figure 8.7.

Each link source activity publishes the transition condition of each outgoing link. In Figure 8.7, 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 8.7, 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 8.7, 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 2.1 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. NÍÑOS 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 updates 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.
Figure 8.8: Process Deployment and Execution Monitor

The variable agent mechanism can always be used, while the 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.

8.1.3 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 \texttt{AGENT\_CTL}, and contain the information that the manager wants to deliver to an agent, and the identifier of the particular agent in the \texttt{agentID} predicate. Activity agents receive the control messages by subscribing to \texttt{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 8.8.

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 \texttt{AGENT\_CTL} publication. Third, the publish/subscribe broker network delivers the \texttt{AGENT\_CTL} publications to the addressed agents. Fourth, the agent extracts the subscription, advertisement, and activity information messages from the \texttt{AGENT\_CTL} message. For instance, Pub1 in Figure 8.8 is an agent control publication wrapping a subscription for the \texttt{while} agent. Finally, the agent processes the messages based on the \texttt{command} field which has three possible values: \texttt{subscribe}, \texttt{advertise} or \texttt{activityinfo}. The \texttt{subscribe} command causes the agent to subscribe to the subscription specified in the \texttt{content} field. The \texttt{advertise} command causes the agent to advertise the advertisement contained in the \texttt{content} field. Subscriptions and advertisements describe the activity dependency of a process. The \texttt{activityinfo} control message contains information needed by the activity agent to execute the activity, such as the Boolean looping condition for a \texttt{while} activity. As a result, 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 accommodating to system requirements. For example, Figure 8.9 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, more resources with which to balance and support greater loads, and redundancy in case of failures. The BPEL process in Figure 8.9 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 8.1.5, 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 8.9) 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 8.9). The scenario in Figure 8.9 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 future work.

8.1.4 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 8.8, and is driven by activity completion events. Execution continues until all the activities defined in the process are finished. The process flow, or dependencies 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
Chapter 8. Case Study: Supporting BPEL

8.1.5 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 NINOS.

NINOS provides a graphical monitoring interface to visualize the network topology, message routing, and distributed process execution, as shown in Figure 8.10. 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 NINOS. Real-time monitoring is simple to achieve in NINOS due to the use of a content-based publish/subscribe infrastructure. The monitor, shown in Figure 8.10, 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 allows 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 8.10 allows an administrator to view, in real-time, the operation of a particular while activity agent in the system including the invocations of the activity.

Enterprise applications also require probing the execution of completed processes, perhaps for auditing or analysis purposes. The PADRES infrastructure supports historic data [48, 55] access using subscriptions that unify the query for past and future events. Along with PADRES’s composite subscriptions feature [52], both executing and previously executed process instances can be correlated and queried. For example, it is possible to
Monitor an activity:
[class,eq,ACTIVITY_STATUS],
[process,eq,Process1],
[activityName,eq,"while"], [IID,isPresent],
[status,isPresent]

Trace a faulty process instance:
[class,eq,ACTIVITY_STATUS],
[activityName,eq,"activity2"], [IID,eq,$X]
[status,eq,"FAILURE"]
&&
[class,eq,ACTIVITY_STATUS],
[activityName,isPresent], [IID,eq,$X]

Pause a process instance:
[class,AGENT_CTL],
[command,PAUSE], [agentID,isPresent],
[process,"PAYROLL"], [IID,p001]

Figure 8.10: PADRES Monitor

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\(^1\) traces the invocations of every activity for those process instances whose activity2 failed. Examining the execution of these instances 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 AGENT_CTL publication in Figure 8.10 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-

\(^1\)We use a composite subscription in this case.
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 8.9, 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 [43], and formalized well-defined semantics for the mobility of activity agents [41, 39].

A larger BPEL example about loan approval is given in Appendix A.

8.1.6 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 8.2.

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 ability the 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 technique 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 fine-grained components make it possible in 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 [86] 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.
8.2 Evaluation

8.2.1 Experimental Setup

We evaluated NIÑOS in a controlled local network with 20 nodes, each of which has 4GB of memory and 1.86GHz CPU. In all the tests, in addition to the deployed activity agents, we add the process management client, and a service request client that invokes process instances. Since there are no accepted benchmarks in this field, we use the delivery service business process described in Figure 8.1.

We compare the centralized, clustered and distributed execution deployments. 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. For the clustered scenario, agents are grouped into two clusters which process service requests concurrently, and the workload is balanced across the two clusters. The two Web services access the system by Web service agents, which are shared between the two clusters. For the distributed scenario, a 30 broker network is deployed on the local network with the agents connecting to the various brokers.

We measure 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 service time, and message size are 500/min, 100 ms, and 512 bytes, respectively.

8.2.2 Request Rate

In this experiment, we vary the process invocation rate, where each invocation generates a process instance, and measure the average execution time and throughput, as shown
Figure 8.11: Performance with different request rates
in Figure 8.11. We see that for lower request rates, the centralized approach offers the best service time, with more than 9% and 20% improvement over the clustered and the distributed approaches. 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, the clustered approach and the distributed approach offer faster service times, with more than 34% and 67% improvement over the centralized scenario at the highest request rate of 6000/min.

The throughput results in Figure 8.11 show that, the distributed and clustered approaches, whose maximum throughput are similar, outperform the centralized one, with a roughly 49% increase in maximum throughput. Interestingly, 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. When the request rate is high, say 1000/min, the clustered and distributed deployments have similar throughput. The reason is that in both approaches, the process needs to access a Web service which has no replica in the clustered approach. The response time of the Web service limits the maximum throughput of both approaches.

### 8.2.3 Web Service Delay

The delay of external Web services affects the execution time and throughput as well. In this experiment, we vary the Web service delay from 20 ms to 2 sec with two different request rates. With a lower request rate of 50/min, in Figure 8.12, we see that, as expected, a longer Web service delay increases the average execution time. When the delay is small, the centralized approach performs the best by avoid extra communication cost in the other two approaches. While when the Web service delay increases, the distributed approach performs the best, with 49% and 70% improvement over the clustered and the centralized scenarios. This is because a large Web service delays result
Figure 8.12: Performance with Different Service Time at Low Request Rate
Figure 8.13: Performance with Different Service Time at High Request Rate
in the execution engine handling 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 higher request rate of 1000/min, as shown in Figure 8.13, the centralized approach performs the worst and the distributed case is the best for all Web service delays. We observed from Figure 8.11 that the throughput in the three approaches reaches the maximum value at the rate of 1000/min. That means more process instances are queueing at activity agents and the Web services. The queueing delay dominates the average execution time and the communication overhead is negligible. In the distributed and clustered approaches, we have more resources and can process the pending activities faster than the centralized one.

### 8.2.4 Message Size

Another factor is the size of the messages, as some processes may need to pass around large data sets. As we vary the message size from 512 bytes to 256 kbytes for different request rates in Figure 8.14 and Figure 8.15, the performance almost is the same in all 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 queueing time dominates the communication overhead.

When the request rate is low (e.g., 50/min), the throughput of all approaches is similar, which is close to the request rate. When the request rate is high (e.g., 500/min), the distributed and clustered approaches maintain throughput that is consistently better than the centralized one, by about 48%. Again, it seems that the communication and processing overheads of traversing a larger broker network is not significant when the message sizes are up to 256 kbytes.
Figure 8.14: Performance with Different Message Sizes at Low Request Rate
Figure 8.15: Performance with Different Message Sizes at High Request Rate

(a) Average Execution Time

(b) Throughput
8.2.5 Parallelism

In order to further investigate the effects of increased parallelism, we compare two processes: one contains many activities with ten parallel branches, as shown in Figure 8.16, and another has the same number activities but only two parallel branches. No Web services are accessed in these processes.

With the highly parallel process, the distributed case offers significant benefits as shown in Figure 8.17(a). When the request rate is less than 100/min, we observe the similar trend as in Figure 8.11, where the distributed case performs worse because of the additional network overhead. When the request rate is high, more activities can be processed in parallel. The distributed case saves the average execution time by 79% over the centralized approach. While for the sequence process, the distributed case has only 42% of improvement over the centralized one. According the the results, the distributed
Figure 8.17: Average Execution Time of Parallel vs. Sequence Processes
approach is, by nature, more suitable for the highly parallel processes and higher service request rate in order to achieve better execution time.
Chapter 9

Conclusions

9.1 Summary and Discussion

This thesis presents a distributed business process execution architecture based on an enhanced publish/subscribe infrastructure. The traditional publish/subscribe model only supports primitive subscriptions, which are not sufficient to support applications such as distributed business process management. Moreover, it does not support the retrieval of data published before a subscription is issued, while the ability to access data from the past and the future in a unified manner is an important and useful requirement in many applications. Routing optimizations and extensions are explored to provide robust and efficient message delivery for high level applications.

Composite subscriptions are naturally supported by the rule-based matching engine in PADRES. In the rule-based matching engine, publications are mapped to facts and subscriptions are mapped to rules. The publication and subscription matching is converted to fact-rule matching. A composite subscription consisting of several primitive subscriptions is mapped to a more expressive rule expression. Each component subscription is part of the rule as a sub-expression. The notification of the composite subscription is a publication message with the set of matched publications as payload.
The proposed PSQL language provides an SQL-based interface to subscribe to historic and future publications. The language fully retains content-based publish/subscribe semantics and features, and can express filtering constraints, aggregation functions, projections, and correlations (joins) across any combination of future and historic data in a manner that preserves publish/subscribe decoupling and anonymity.

Content-based routing in a general overlay improves the scalability and robustness of publish/subscribe systems by offering routing path alternatives. Our approach retains the original publish/subscribe interface and matching algorithms so it may be easily integrated in existing publish/subscribe systems. It also minimizes redundant traffic induced by cycles in the overlay and reduces message routing delay. As well novel protocols allow publications to select “optimal” paths to matching subscribers and composite subscriptions can be routed to the “best” event detection locations in order to satisfy potential quality of service constraints at the application level.

Experiments in both wide area PlanetLab and controlled local environments confirm the benefits of the dynamic publication routing algorithm. Publication end to end routing is about 20% faster, stabilizes sooner after a burst, and is able to route around certain failures in the network.

An architecture to evaluate PSQL subscriptions is presented and employs a set of databases to store publications and respond to historic subscriptions. Concepts from distributed databases are used but the publish/subscribe model differs sufficiently from the relational model to preclude a direct transfer of the ideas. The database architecture supports a range of distribution and replication strategies with the ability to arbitrarily assign portions of the data space to one or more databases. An algorithm is presented to uniformly partition the data space, and a synchronization protocol is developed to ensure replica consistency among replicas.

To optimize the evaluation of PSQL queries, various subscription routing protocols are presented. The simple routing policy is to separate a composite subscription into primi-
ative subscriptions at the first broker which receives the subscription. The topology-based routing policy is to decompose a composite subscription only if publications matching its components come from different brokers. The adaptive routing policy based on the cost model determines the least costly location in the overlay to compute composite subscription correlations. The algorithm is a greedy algorithm based on local knowledge reaching a global optimal solution. Our experiments show that the network traffic is reduced by using composite subscriptions. The number of messages received by clients is reduced by using notification messages. The distributed routing policies reduces the number of publications routed in the broker network as well. Furthermore, detecting composite events within the publish/subscribe system allows subscribers to share the detection results. Since no detection module is needed on the subscriber side, subscribers can be lightweight.

Evaluations demonstrate the tradeoffs of several partition assignment and replication policies. Assigning all partitions to every database imposes the greatest publication storage traffic but cheapest historic subscription overhead, whereas a fully distributed partitioning results in the opposite tradeoff. The relative bias of writes (publications) and reads (subscriptions) in an application will determine the appropriate replication policy. In terms of delivering notifications, however, full replication takes longer than with complete partitioning because the latter is able to parallelize subscription processing across several databases, even though the databases are further away. That is, parallelization of subscription processing benefits much more than locality of databases. Furthermore, partitioning is able to start delivering notifications sooner than full replication, again due to parallelism. Adaptive subscription routing successfully determines efficient locations in the network to evaluate composite subscription correlations, and dynamic routing is able to react to changes in workloads and readjust the correlation evaluation locations, reducing network traffic even further.

We also extend the language model from predicate-based languages to XML-based
languages. In the dissemination network, publishers’ DTD files are transformed into advertisements expressed using XPath-like expressions. An advertisement creates a spanning tree rooted at the publisher. Subscribers specify XPath filters which are forwarded along the reverse paths of this tree for intersecting advertisements. XML documents from publishers are forwarded along these routing paths to subscribers with matching XPEs. By defining and exploiting covering and merging relations for XPEs, a compact routing table results. We perform a detailed experimental evaluation of our approach on an overlay network comprised of 127 XML routers deployed over a cluster with 25 nodes and deployed on PlanetLab. Our experimental results demonstrate the effectiveness of the approach by reducing the routing table size by up to 90% and improving the routing time by roughly 85% in the most favorable cases. Our experiments suggest that the scalability of the system is improved by applying advertisement-based routing, covering, and merging techniques for routing XML documents in a data dissemination network. In large-scale publish/subscribe systems, covering and merging techniques improve the publication matching efficiency by reducing the routing table size and decrease the network traffic overhead by eliminating redundant subscriptions routed within the network.

With composite subscriptions, historic data access and the optimizations of dynamic publication routing, covering and merging, we propose a distributed business process execution architecture, based on a publish/subscribe infrastructure, using light-weight activity agents to carry out 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 even-driven and loosely coupled nature of 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.

9.2 Future Work

We plan to extend our work in several directions. First, while the subscription and publication workload changes, for the historic data access module, it maybe worthwhile to repartition the publication space dynamically online. We studied horizontal partitioning in this work, other partitioning approaches, such as vertical partitioning, might be interesting to explore. We proposed several cost models that quantify the cost of retrieving data from various partitions, and heuristics that minimize these cost functions. Further study is needed to evaluate these cost models and to characterize what metrics effect the performance under what scenario, etc. Second, we plan to extend the PSQL language to manipulate running instances of business processes, explore the general patterns in large business processes and workflows, and provide more process management functionality, such as versioning management, online and offline process monitoring. Third, it worths studying the movement of activity agents in order to satisfy certain goals or constraints. For example, the average execution time may be minimized by moving tightly coupled activity agents close to one another. To monitor whether an application is providing (and receiving) the desired performance, it is common for businesses to define service level agreements (SLAs) on large-scale enterprise applications. Monitoring SLAs
in NIÑOS is another interesting research direction to further explore. Moreover, it is also interesting to study more advanced techniques for the validation of BPEL process specifications, such as model checking and simulations for process execution debugging in the distributed environment. Last, we would like to explore more experiments with larger business processes and broker topologies.
Appendix A

BPEL Example: Loan Approval

Consider a loan approval BPEL process in Figure A.1. 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 to 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 A.1 is mapped to a NIÑOS agent. Table A.1 details the advertisements and subscriptions issued by each agent, as well the publications for a sample run of the process.

<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],</td>
<td>[class,TRIGGER],</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[process,eq,&quot;loan&quot;],</td>
<td>[process,&quot;loan&quot;],</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[IID,isPresent]</td>
<td>[IID,&quot;g001&quot;], &lt;&lt;CLAIM&gt;&gt;</td>
</tr>
</tbody>
</table>
### Appendix A. BPEL Example: Loan Approval

#### Receive1

<table>
<thead>
<tr>
<th>Event</th>
<th>Class</th>
<th>Eq</th>
<th>Process</th>
<th>Activity Name</th>
<th>IID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>class</td>
<td>eq</td>
<td>TRIGGER</td>
<td>loan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>process</td>
<td>eq</td>
<td>loan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>IID</td>
<td>isPresent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Assign1

<table>
<thead>
<tr>
<th>Event</th>
<th>Class</th>
<th>Eq</th>
<th>Process</th>
<th>Activity Name</th>
<th>IID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>class</td>
<td>eq</td>
<td>ACTIVITY_STATUS</td>
<td>loan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>process</td>
<td>eq</td>
<td>loan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>activityName</td>
<td>eq</td>
<td>receive1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>IID</td>
<td>isPresent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>status</td>
<td>isPresent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Invoke1

<table>
<thead>
<tr>
<th>Event</th>
<th>Class</th>
<th>Eq</th>
<th>Process</th>
<th>Activity Name</th>
<th>IID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>class</td>
<td>eq</td>
<td>ACTIVITY_STATUS</td>
<td>loan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>process</td>
<td>eq</td>
<td>loan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>activityName</td>
<td>eq</td>
<td>receive1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>IID</td>
<td>eq</td>
<td>$X$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>status</td>
<td>eq</td>
<td>SUCCESS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CLAIM**

**RESULT**

---

**Flow1**

<table>
<thead>
<tr>
<th>Event</th>
<th>Class</th>
<th>Eq</th>
<th>Process</th>
<th>Activity Name</th>
<th>IID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>class</td>
<td>eq</td>
<td>ACTIVITY_STATUS</td>
<td>loan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>process</td>
<td>eq</td>
<td>loan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>activityName</td>
<td>eq</td>
<td>receive1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>IID</td>
<td>eq</td>
<td>$X$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>status</td>
<td>eq</td>
<td>SUCCESS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix A. BPEL Example: Loan Approval

<table>
<thead>
<tr>
<th>wsA</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[class, eq, ACTIVITY_STATUS],</code></td>
<td><code>[class, eq, ACTIVITY_STATUS],</code></td>
<td><code>[class, ACTIVITY_STATUS],</code></td>
<td></td>
</tr>
<tr>
<td><code>[process, eq, &quot;loan&quot;],</code></td>
<td><code>[activityName, eq, &quot;wsA&quot;],</code></td>
<td><code>[process, &quot;loan&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[activityName, eq, &quot;invoke1&quot;],</code></td>
<td><code>[IID, isPresent],</code></td>
<td><code>[activityName, &quot;wsA&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[IID, eq, $X],</code></td>
<td><code>[status, isPresent]</code></td>
<td><code>[IID, eq, &quot;g001&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[status, eq, &quot;SUCCESS&quot;]</code></td>
<td></td>
<td><code>[status, eq, &quot;SUCCESS&quot;],</code></td>
<td></td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td></td>
<td><code>&lt;&lt;RESULT&gt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>[class, eq, VARIABLE_UPDATE],</code></td>
<td><code>[class, eq, VARIABLE_UPDATE],</code></td>
<td><code>[class, VARIABLE_UPDATE],</code></td>
<td></td>
</tr>
<tr>
<td><code>[process, eq, &quot;loan&quot;],</code></td>
<td><code>[activityName, eq, &quot;receive1&quot;],</code></td>
<td><code>[process, &quot;loan&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[activityName, eq, &quot;receive1&quot;],</code></td>
<td><code>[IID, eq, $X],</code></td>
<td><code>[activityName, &quot;receive2&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[IID, eq, $X],</code></td>
<td><code>[variableName, eq, &quot;claim&quot;]</code></td>
<td><code>[IID, isPresent],</code></td>
<td></td>
</tr>
<tr>
<td><code>[variableName, eq, &quot;claim&quot;]</code></td>
<td></td>
<td><code>[variableName, eq, &quot;reportA&quot;],</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;&lt;RESULT&gt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td>receive2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>[class, eq, ACTIVITY_STATUS],</code></td>
<td><code>[class, eq, ACTIVITY_STATUS],</code></td>
<td><code>[class, ACTIVITY_STATUS],</code></td>
<td></td>
</tr>
<tr>
<td><code>[process, eq, &quot;loan&quot;],</code></td>
<td><code>[activityName, eq, &quot;receive2&quot;],</code></td>
<td><code>[process, &quot;loan&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[activityName, eq, &quot;wsA&quot;],</code></td>
<td><code>[IID, isPresent],</code></td>
<td><code>[activityName, &quot;receive2&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[IID, isPresent],</code></td>
<td><code>[status, isPresent]</code></td>
<td><code>[IID, eq, &quot;g001&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[status, eq, &quot;SUCCESS&quot;]</code></td>
<td></td>
<td><code>[status, eq, &quot;SUCCESS&quot;],</code></td>
<td></td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td></td>
<td><code>&lt;&lt;RESULT&gt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>[class, eq, VARIABLE_UPDATE],</code></td>
<td><code>[class, eq, VARIABLE_UPDATE],</code></td>
<td><code>[class, VARIABLE_UPDATE],</code></td>
<td></td>
</tr>
<tr>
<td><code>[process, eq, &quot;loan&quot;],</code></td>
<td><code>[activityName, eq, &quot;receive2&quot;],</code></td>
<td><code>[process, &quot;loan&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[activityName, eq, &quot;receive2&quot;],</code></td>
<td><code>[IID, isPresent],</code></td>
<td><code>[activityName, &quot;receive2&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[IID, isPresent],</code></td>
<td><code>[variableName, eq, &quot;reportA&quot;]</code></td>
<td><code>[IID, eq, &quot;g001&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[variableName, eq, &quot;reportA&quot;]</code></td>
<td></td>
<td><code>[variableName, eq, &quot;scoreA&quot;]</code></td>
<td></td>
</tr>
<tr>
<td>assign2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>[class, eq, ACTIVITY_STATUS],</code></td>
<td><code>[class, eq, ACTIVITY_STATUS],</code></td>
<td><code>[class, ACTIVITY_STATUS],</code></td>
<td></td>
</tr>
<tr>
<td><code>[process, eq, &quot;loan&quot;],</code></td>
<td><code>[activityName, eq, &quot;assign2&quot;],</code></td>
<td><code>[process, &quot;loan&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[activityName, eq, &quot;receive2&quot;],</code></td>
<td><code>[IID, isPresent],</code></td>
<td><code>[activityName, &quot;assign2&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[IID, eq, $X],</code></td>
<td><code>[status, isPresent]</code></td>
<td><code>[IID, eq, &quot;g001&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[status, eq, &quot;SUCCESS&quot;]</code></td>
<td></td>
<td><code>[status, &quot;SUCCESS&quot;],</code></td>
<td></td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td></td>
<td><code>&lt;&lt;RESULT&gt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>[class, eq, VARIABLE_UPDATE],</code></td>
<td><code>[class, eq, VARIABLE_UPDATE],</code></td>
<td><code>[class, VARIABLE_UPDATE],</code></td>
<td></td>
</tr>
<tr>
<td><code>[process, eq, &quot;loan&quot;],</code></td>
<td><code>[activityName, eq, &quot;assign2&quot;],</code></td>
<td><code>[process, &quot;loan&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[activityName, eq, &quot;assign2&quot;],</code></td>
<td><code>[IID, isPresent],</code></td>
<td><code>[activityName, &quot;assign2&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[IID, eq, $X],</code></td>
<td><code>[variableName, eq, &quot;scoreA&quot;]</code></td>
<td><code>[IID, eq, &quot;g001&quot;],</code></td>
<td></td>
</tr>
<tr>
<td><code>[variableName, eq, &quot;reportA&quot;]</code></td>
<td></td>
<td><code>[variableName, &quot;scoreA&quot;],</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;&lt;APPROVAL&gt;&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
switch1
[\text{class}, \text{eq}, \text{ACTIVITY STATUS}],
[\text{process}, \text{eq}, \text{"loan"}],
[\text{activityName}, \text{eq}, \text{"assign2"}],
[IID, \text{eq}, \$,X$],
[status, \text{eq}, \text{"SUCCESS"}] &&
[\text{class}, \text{eq}, \text{VARIABLE UPDATE}],
[\text{process}, \text{eq}, \text{"loan"}],
[\text{activityName}, \text{eq}, \text{"assign2"}],
[IID, \text{eq}, \$,X$],
[variableName, \text{eq}, \text{"scoreA"}] &&
[\text{class}, \text{eq}, \text{ACTIVITY STATUS}],
[\text{process}, \text{eq}, \text{"loan"}],
[\text{activityName}, \text{eq}, \text{"assign3"}],
[IID, \text{eq}, \$,X$],
[status, \text{eq}, \text{"SUCCESS"}] &&
[\text{class}, \text{eq}, \text{VARIABLE UPDATE}],
[\text{process}, \text{eq}, \text{"loan"}],
[\text{activityName}, \text{eq}, \text{"assign3"}],
[IID, \text{eq}, \$,X$],
[variableName, \text{eq}, \text{"scoreB"}]

assign4
[\text{class}, \text{eq}, \text{SWITCH CASE TRIGGER}],
[\text{process}, \text{eq}, \text{"loan"}],
[\text{activityName}, \text{eq}, \text{"switch1"}],
[IID, \text{isPresent}],
[triggeredActivity, \text{eq}, \text{"assign4"}] &&
[\text{class}, \text{eq}, \text{ACTIVITY STATUS}],
[\text{process}, \text{eq}, \text{"loan"}],
[\text{activityName}, \text{eq}, \text{"assign4"}],
[IID, \text{isPresent}],
[status, \text{isPresent}] &&
[\text{class}, \text{eq}, \text{VARIABLE UPDATE}],
[\text{process}, \text{eq}, \text{"loan"}],
[\text{activityName}, \text{eq}, \text{"assign4"}],
[IID, \text{isPresent}],
[variableName, \text{eq}, \text{"approval"}]

[\text{class,SWITCH_CASE_TRIGGER}],
[\text{process,}\text{"loan"}],
[\text{activityName,} \text{"switch1"}],
[IID,\text{"g001"}],
[\text{triggeredActivity,} \text{"assign4"}]

[\text{class,ACTIVITY_STATUS}],
[\text{process,}\text{"loan"}],
[\text{activityName,} \text{"assign4"}],
[IID,\text{"g001"}],
[\text{status,} \text{"SUCCESS"}]

<<YES>>
Some of the agents in the parallel branches of the process, such as the invoke2 and receive3 activities, are omitted from Table A.1. Their messages would correspond to the messages issued by their corresponding activities in the other branch.

Although there is a flow activity in the BPEL process in Figure A.1, there is no corresponding agent. Instead, the first activity in each branch of the flow are triggered when the final activity before flow 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 A.1, however, to illustrate what its message would look like.
Appendix A. BPEL Example: Loan Approval

Figure A.1: Example Loan Application Process
Bibliography


[47] V. Kumar, Z. Cai, B. F. Cooper, G. Eisenhauer, K. Schwan, M. Mansour, B. Seshasayee, and P. Widener. Implementing diverse messaging models with self-


