

AN APPROACH TO MANAGED CLUSTERING FOR KNOWLEDGE- BASED NETWORKS

A thesis submitted to the
University of Dublin, Trinity College
for the degree of
Doctor of Philosophy

Dominic Hugh Jones, B.Sc. (Hons), M.Sc.
Knowledge and Data Engineering Group (KDEG)
School of Computer Science and Statistics
Trinity College, Dublin,
Dominic.Jones@cs.tcd.ie

Supervised by Dr. David Lewis
Co-Supervised by: Dr. Declan O'Sullivan

Declaration and Online Access

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Dominic Hugh Jones

April 2013

Acknowledgements

I would first like to thank Dr. David Lewis for providing me with this opportunity to pursue the degree of Doctor of Philosophy. Dr. Lewis has been supportive, critical, encouraging and has become both a friend and mentor over my time in KDEG, it has been an honour to work with Dr. Lewis.

Dr. Lewis has been able to offer me funding through Science Foundation Ireland under Grant No 05/RFP/CMS014 (Mecon) and Grant number 07/CE/I1142 (CNGL). For this I am extremely appreciative.

It has been a privilege to have the co-supervision of Dr. O'Sullivan who offered me advice and support that has been invaluable over the years. Special thanks to Dr. John Keeney, Dr. Alexander O'Connor, Dr. Seamus Lawless, Dr. Owen Conlan and Mrs. Mary Sharp who have supported and encouraged me throughout the years and made the difference to many a day.

It was Ms. Marie Carroll who guided me through my undergraduate study, provided endless cups of tea and was there when needed. Marie will always be a great friend, who started me on my path through University level education and is herself an asset to education.

However most importantly I would like to thank my parents Jan and Hughie Jones as well as my de-facto American Mom Dr. P. Jane Gale. Without their combined support this thesis could easily have disappeared during its final year in which they picked me up, stood by me, guided me and had faith in me.

“Insanity is doing the same thing, over and over again, but expecting different results.”

- Possibly Albert Einstein, possibly not.

Abstract

In publish/subscribe networks the capacity of the delivery network can easily be exceeded as the volume and range of content increases. This is known as the flooding problem. One solution to this problem is to group publishers and subscribers with similar interests into clusters.

A publish/subscribe network typically operates across a broker overlay. The work described in this thesis offers a novel solution to the flooding problem based on reducing the number of brokers between related publishers and subscribers, thus reducing the routing message volume. With fewer brokers between publisher and subscriber, fewer hops are required to route each message and ultimately the network is more efficient, with less resources required to deliver the same number of messages.

This thesis describes the design, implementation and evaluation of a prototype system in which a Semantic-based Publish/Subscribe network is used to cluster publishers of content with the message brokers that are responsible for routing content to subscribers with related interests. The objective of this clustering is to reduce the number of hops required in the routing process.

This contribution has been implemented in a prototype, KBNCluster, as part of this thesis and provides a platform for the clustering of KBNImpl, which has been shown to reduce the hop count required to deliver publications to subscribers and thus decrease the time taken to deliver publications.

A further contribution of this work is the automatic calculation, placement and re-clustering of publishers and subscribers around their interests using a Policy-based Network Management approach. This contribution is evaluated in terms of management data collection costs, storage and policy execution and is shown to scale efficiently.

This work is novel in the way in which semantic interests are calculated from a client's publication or subscription, using a semantic map of concepts to calculate a single semantic point deemed representative of the client. An additional source of novelty in this work is in employing the network's Semantic-based Publish/Subscribe capability for the delivery of management messages to the management system using a push-based approach in contrast to more standard pull-based architectures.

Table of Contents

Declaration and Online Access.....	II
Acknowledgements.....	III
Abstract.....	IV
Table of Contents.....	V
List Of Figures.....	XI
List Of Tables.....	XIII
List Of Code Examples.....	XIV
Abbreviations.....	XV
1 INTRODUCTION.....	1
1.1 Research Question.....	5
1.2 Research Objectives.....	5
1.3 Methodological Approach Taken.....	6
1.4 Evaluation Findings.....	6
1.5 Thesis Contribution.....	8
1.6 Selected publications.....	9
1.7 Thesis Outline.....	10
2 BACKGROUND.....	11
2.1 Key Terms.....	11
2.2 Semantics.....	12
2.2.1 RDF and Ontologies.....	12
2.3 Publish/Subscribe Middleware.....	15
2.3.1 Topic-based Publish / Subscribe:.....	16
2.3.2 Content-based Publish / Subscribe.....	16
2.3.3 Knowledge-based Networks in comparison to CBNs.....	17
2.4 Policy-based Network Management (PBNM).....	20
2.5 Conclusion.....	21
3 STATE OF THE ART.....	22
3.1 Content-based Networks.....	22
3.1.1 Siena.....	22
3.1.2 Hermes.....	23
3.1.3 Gryphon.....	24

3.1.4	Elvin	24
3.1.5	Analysis	25
3.1.5.1	<i>Advantages of CBNs</i>	25
3.1.5.2	<i>Disadvantages of CBNs</i>	25
3.2	Semantic-based Publish/Subscribe (SBPS)	26
3.2.1	Knowledge-based Networks	26
3.2.1.1	<i>KBNImpl</i>	26
3.2.1.2	<i>KBNMap</i>	26
3.2.1.3	<i>KBNCluster</i>	27
3.2.2	A semantic Infosphere	27
3.2.3	Semantic Toronto Publish/Subscribe System (S-ToPSS)	28
3.2.4	Graphed Toronto Publish/Subscribe System (G-ToPSS)	29
3.2.5	Semantic Message Middleware for publish/subscribe networks (SMOM)	29
3.2.6	An ontology-Based publish/subscribe System (OPS)	30
3.2.7	Designing semantic Publish Subscribe networks Using Super-Peers (SPS-SP)	30
3.2.8	iBroker	31
3.2.9	Analysis	31
3.2.9.1	<i>Advantages of SBPS</i>	31
3.2.9.2	<i>Disadvantages of SBPS</i>	32
3.3	Publish / Subscribe clustering Techniques	33
3.3.1	Topic-based Publish/Subscribe Clustering	34
3.3.1.1	<i>Boosting topic-based publish-subscribe systems with dynamic clustering (Tamara)</i>	34
3.3.1.2	<i>Scribe: a large-scale and decentralized application-level multicast infrastructure</i>	34
3.3.1.3	<i>Data Aware Multicast (daMulticast)</i>	35
3.3.1.4	<i>Topic-based Event Routing for peer-to-peer Architectures (TERA)</i>	35
3.3.2	Content-based Publish/Subscribe Clustering	36
3.3.2.1	<i>Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks</i>	36
3.3.2.2	<i>Efficient Publish Subscribe through a Self-Organizing broker Overlay and its Application to SIENA</i>	37
3.3.3	Conclusion	37
3.4	Publish/Subscribe Classification II	38
3.5	Research Challenges gathered from the SoA	39
3.5.1	Design Ideas gathered from the SoA	40
3.5.1.1	<i>Subscription Regionalism</i>	40
3.5.1.2	<i>Ontology Partitioning</i>	41
3.5.1.3	<i>Ontological Change</i>	42
3.6	Conclusion	43

4	DESIGN.....	44
4.1	Introduction	44
4.1.1	Extended and New Technology.....	46
4.2	High Level Design.....	47
4.2.1	System Architecture	47
4.2.2	Design Overview	48
4.2.2.1	<i>Managed Overlay Design Overview</i>	48
4.2.2.2	<i>Trigger Broker Design Overview</i>	48
4.2.2.3	<i>Policy Server Design Overview</i>	48
4.2.3	Clustering Design Assumptions and Scope.....	49
4.2.3.1	<i>Publisher and Subscriber Design Assumptions</i>	49
4.2.3.2	<i>Design Assumptions around ontological clustering</i>	49
4.3	Clustering Process	51
4.3.1	The Medoid	51
4.3.2	Taxonomical Approach to Cluster Creation.....	53
4.3.3	Creating an Ontological A* Map.....	55
4.3.3.1	<i>Extracting Semantic Elements from Subscriptions</i>	56
4.3.3.2	<i>Extracting Semantic Elements from Publications</i>	56
4.3.4	Medoid Calculation	57
4.3.4.1	<i>Example Medoid Calculation</i>	58
4.3.4.2	<i>Summary: Medoid Calculation</i>	58
4.3.5	Cluster Placement.....	59
4.3.6	Re-clustering.....	60
4.4	Trigger Broker.....	62
4.4.1	Subscriptions to Trigger Broker	63
4.4.2	Publications Received by the Trigger Broker.....	64
4.4.3	Interactions between KBNImpl Publishers and the Trigger Broker.....	66
4.5	Policy Server	67
4.5.1	MIB/MO Design.....	68
4.5.2	Policies	69
4.5.3	Actionable events	70
4.6	Key Design Characteristics and Conclusion	72
5	IMPLEMENTATION.....	74
5.1	Introduction	74
5.2	Technology Selection	75
5.2.1	Core Technologies:.....	75
5.2.2	Development Tools:	75

5.2.3	Messaging Mechanisms:	76
5.3	Order of Operation	77
5.4	Communication Flow	79
5.4.1	Overall Communication	80
5.4.2	Broker Focused Communication	82
5.4.3	Publisher Focused Communication	83
5.4.4	Subscriber Focused Communication	85
5.4.5	Conclusions – Communication Flow	86
5.5	Trigger Broker	87
5.5.1	Trigger Broker – Start-up Configuration	87
5.5.2	Trigger Broker – Main Classes	87
5.5.3	Trigger Broker - Summary of Key Characteristics	89
5.6	Policy Server	90
5.6.1	Policy Server - Start-up Configuration	90
5.6.2	Main Classes	91
5.6.3	MIB/MO Implementation	92
5.6.4	Clustering Process	93
5.6.4.1	<i>Cluster Partitioning</i>	93
5.6.4.2	<i>Overlaying an Ontology onto Brokers</i>	94
5.6.4.3	<i>Calculating Client Cluster Placement</i>	95
5.6.5	Summary of Key Characteristics	95
5.7	Starting Brokers, Publisher and Subscriber	96
5.7.1	Starting a Broker	96
5.7.2	Starting a Publisher	97
5.7.3	Starting a Subscriber	99
5.8	Conclusions and Summary of Technical Discussions	100
6	EVALUATION	101
6.1	Experimental Setup	103
6.1.1	Ontologies	103
6.1.2	Platforms	104
6.1.3	Metrics	105
6.1.4	Sensitivity Analysis	106
6.2	Static Approach to Clustering	109
6.2.1	Introduction	109
6.2.2	Experimental Metrics	109
6.2.3	Experimental Setup	110

6.2.4	Results	110
6.2.5	Conclusion	112
6.3	KBNImpl Operational Costs	113
6.3.1	Operator Usage	113
6.3.2	Subscription Tree Search Time	115
6.3.3	Hop Count Experiments	117
6.3.3.1	<i>Hop Occurrences</i>	117
6.3.3.2	<i>Hop Count Timing</i>	119
6.3.4	Conclusion	120
6.4	Costs Associated with Dynamic Clustering	121
6.4.1	Management Data Storage & Policy Execution Costs	122
6.4.1.1	<i>Memory Footprint</i>	122
6.4.1.2	<i>Policy Execution Timing</i>	123
6.4.1.3	<i>Conclusion</i>	126
6.4.2	Data Collection Costs	126
6.4.2.1	<i>Management Method 1</i>	127
6.4.2.2	<i>Management Method 2</i>	128
6.4.2.3	<i>Standard Error Calculation</i>	129
6.4.2.4	<i>Subscription Processing Times</i>	130
6.4.2.5	<i>Publication Processing Times</i>	131
6.4.2.6	<i>Pub-to-Sub Delivery Times</i>	133
6.4.2.7	<i>Conclusion</i>	134
6.4.3	Mobility Costs	136
6.4.3.1	<i>Moving Broker & Moving All Subscribers</i>	137
6.5	Dynamic Clustering Evaluation	139
6.5.1	Experimental Metrics	140
6.5.2	Subscription Tree Size	140
6.5.3	Hop Count in Delivery	142
6.5.4	Re-Clustering	144
6.5.5	Conclusion	146
6.6	Overall Conclusion	147
7	Conclusion	149
7.1	Objectives and Achievements	149
7.1.1	Research Objective 1	150
7.1.2	Research Objective 2	151
7.1.3	Research Objective 3	151
7.2	Contributions	152

7.2.1	Major Contribution	152
7.2.2	Minor Contribution.....	153
7.2.3	Additional Supporting Publications:	154
7.3	Future Work	155
7.3.1	Subscriber Re-clustering.....	155
7.3.2	Load Balancing of Clusters	155
7.3.3	The trigger broker as a management event monitoring system.....	156
7.3.4	Multiple Policy Servers	156
7.4	Final Remarks.....	157
Bibliography		158
Appendices		161
Appendix A		161
Appendix B		169
Appendix C		178
Appendix D		182
Appendix E		183

List Of Figures

Figure 1: Example Ontology.....	13
Figure 2: Classification of Publish/Subscribe Systems	15
Figure 3: Classification of Publish/Subscribe Systems II.....	38
Figure 4: High Level Component Architecture	47
Figure 5: Sample Clustered Ontology.....	53
Figure 6: A* Ontological Map.....	55
Figure 7: Sample ontology graphed using A* algorithm.....	56
Figure 8: Example Medoid Calculation.....	58
Figure 9: Trigger Broker, High Level Overview	62
Figure 10: Trigger Broker Ontology.....	63
Figure 11: Policy Server, High Level Architecture	67
Figure 12: Components of KBNCluster	77
Figure 13: Overall Communication Mechanisms.....	80
Figure 14: Broker Based Communication	82
Figure 15: Publisher based communication.....	83
Figure 16: Subscriber based Communication	85
Figure 17: UML Class Diagrams for Broker, Publisher and Subscriber	92
Figure 18: Broker Hierarchy.....	110
Figure 19: Root Subscription Tree Size on the Master Node (shown in Figure 18)	111
Figure 20: Unique Subscriptions on a Broker on Level 3 (shown in Figure 18).....	111
Figure 21: Example Topology, Operator Search Costs	114
Figure 22: Subscription tree search times (Semantic & Non Semantic)	116
Figure 23: Spread of Hop Counts in Delivered Notifications, across various cluster topologies	118
Figure 24: Average Message Delivery Timing, Semantic.....	119
Figure 25: Average Message Delivery Times, Non Semantic.....	120
Figure 26: Memory Usage vs. MO Created.....	123
Figure 27: Time Taken to execute a number of MIBS against various Policies	125

Figure 28: Average Subscription Processing Times (ms).....	131
Figure 29: Average Publication Processing Times (ms).....	132
Figure 30: End-to-End Delivery Time (ms).....	134
Figure 31: Moving Broker & All Subs and Moving All Subs Individually	138
Figure 32: Dynamic Clustering - Server Topology	139
Figure 33: Root Subscription Tree Size (Clustered and un-clustered)	141
Figure 34: Hop Count - Clustered and Un-Clustered Topology.....	143
Figure 35: Mean Hop Count in Delivery for Each Subscriber	145

List Of Tables

Table 1: Example RDF Triple.....	12
Table 2: Example Filter Constraint.....	16
Table 3: Example Publication.....	16
Table 4: Example Subscription.....	17
Table 5: Classification of Topic, Content and Knowledge-based Network	18
Table 6: Siena CBN Operators, Types and Symbols.....	19
Table 7: KBNImpl KBN Operators, Types and Symbols.....	19
Table 8: Bag Operators, Types and Symbols.....	19
Table 9: Broker, Subscriber & Publisher MO Key Elements	68
Table 10: Test Ontology Characteristics.....	104
Table 11: Matrix of Experimental Metrics	107
Table 12: Experimental set-up metrics used in experiments	109
Table 13: Experimental set-up metrics used in experiments	113
Table 14: Example Subscriptions and Matching Publications	113
Table 15: KBNImpl Operator Costs – End to end mean delivery times per match (ms)	114
Table 16: Experimental set-up metrics used in experiments	115
Table 17: Experimental set-up metrics used in experiments	117
Table 18: Experimental set-up metrics used in experiments	130
Table 19: Standard Error Subscription Processing Times (ms).....	131
Table 20: Experimental set-up metrics used in experiments	132
Table 21: Standard Error Publication Processing Times (ms).....	132
Table 22: Experimental set-up metrics used in experiments	133
Table 23: Standard Error End-to-End Delivery Times (ms).....	134
Table 24: Experimental set-up metrics used in experiments	137
Table 25: Experimental Data	138
Table 26: Experimental set-up metrics used in experiments	140

List Of Code Examples

Code Example 1: Example Managerial Subscription II	63
Code Example 2: Example Managerial Publication II	64
Code Example 3: Example Medoid Message	64
Code Example 4: Example Broker Info Message.....	64
Code Example 5: Example Subscriber Info Message (Publication).....	65
Code Example 6: Example Publisher Info Message (Publication).....	65
Code Example 7: Example Subscriber Notification Report (Publication)	65
Code Example 8: Starting the Trigger Broker	87
Code Example 9: Starting the Policy Server.....	90
Code Example 10: Starting the Top Level Master Broker	96
Code Example 11: Starting a Sub Broker	96
Code Example 12: Example Publishing Client.....	97
Code Example 13: Example Subscriber Client.....	99
Code Example 14: Example Semantic Subscription used within this section.....	116
Code Example 15: Example Semantic Publications used within this section	116
Code Example 16: Calculating Memory Usage.....	122
Code Example 17: Request Medoid	127
Code Example 18: Reduce down to N clusters.....	128
Code Example 19: Request Medoid Info.....	128
Code Example 20: Example Subscriptions used within the Section	136
Code Example 21: Dynamic Clustering, example subscriptions.....	140
Code Example 22: Dynamic Clustering, example publications	140
Code Example 23: Re-cluster Messages received by Subscribers.....	146

Abbreviations

AC	Attribute Constraint
API	Application Programming Interface
AV	Attribute Value
BIM	Broker Information Message
CBN	Content-based Network
DEBS	Distributed Event-based Systems
DHT	Distributed Hash Table
DPS	Distributed-Publish/Subscribe
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
JMS	Java Messaging Service
KBN	Knowledge-based Network
MIB	Management Information Base
MO	Managed Object
OAEI	Ontology Alignment Evaluation Initiative
OWL	Ontology Web Language
PAM	Partitioning Around Medoids
PBNM	Policy-based Network Management
PIM	Publisher Information Message
RBAC	Role-based Access Control
RDF	Resource Description Framework
RSS	Really Simple Syndication
SBPS	Semantic-based Publish/Subscribe
SIM	Subscriber Information Message
SoA	State of the Art
SPARQL	SPARQL Protocol and RDF Query Language
TBN	Topic-based Network
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
XML	Extensible Mark-up Language

1 INTRODUCTION

Publish/subscribe networks are made up of client content producers (publishers) and client consumers (subscribers) both attached to a network of message brokers/routers. Currently clients make an un-constrained choice as to the broker to connect to, often leading to an overloading of certain popular brokers in the network, resulting in performance degradation. This thesis is designed to improve the scalability of Semantic-based Publish/Subscribe (SBPS) networks by placing clients interested in similar content around similar brokers in the network.

These brokers asynchronously route messages from publisher to interested subscribers based on matches between publication and subscription. Publications are messages that may be of interest to subscribers, where those interests have been expressed in a subscription. Subscriptions are expressions of subscribers interests, often formed from a taxonomy of types/topics or from a set of one or more filter constraints defining the content they wish to receive from publishers. A broker maintains a set of these subscriptions, and applies their subscription constraints to publications as they arrive. If a publication is matched against a subscription, then it is forwarded to the associated subscriber.

Centralised publish/subscribe networks operate using a single broker, whereas distributed publish/subscribe networks operate across a network of inter-connected message brokers. In distributed publish/subscribe networks, messages are usually routed from publisher to relevant subscribers according to a routing table derived from subscription filters. This offers performance efficiencies, as messages are directed only to interested subscribers, rather than using an approach which floods the network in the search for all possible interested publishers.

The main differentiators between classes of publish/subscribe systems are the formats, types, values, ranges and filter operators supported in specifying subscriptions and publications. For example highly expressive filters can be argued to more accurately represent subscribers interests through a greater number of filter constructs. Therefore asserting that the more expressive the subscription filter, the greater the ability of the subscriber to accurately express their interests. In increasing order of expressivity, the three main classes of publish/subscribe network are:

1. Type/Topic-based publish/subscribe (TBN): Publications are tagged with a topic or type. Types/topics are often formed into taxonomies with parent/child relationships existing between type/topic values. Subscribers use the operators: *sub*, *super*, *not/exact* against types/topics. For example a subscription may be made to “every *sub* topic of Football” or “every *super* topic of Manchester United.” Matching occurs using only the header of the publication, which contains the type/topic, and the stored subscription, using the specified operator. (TIBCO [1] is a widely know example of topic-based publish/subscribe.)

2. Content-based publish/subscribe (CBN): Publications contain a set of one or more typed values, whereas subscriptions are a set of value-based filters that are applied to the body of the publication's contents (Siena [2] being an example of a CBN.)
3. Semantic-based publish/subscribe (SBPS): Similar to content-based, but where both publications and subscriptions may have semantically enhanced operators, types and values based on a semantic model or ontology [3]. The use of the term SBPS is first referred to by Guo in [4]. (Examples of such SBPS can be found in [5-9] an example being Knowledge-based Networks (KBN) [9].)

Type / Topic-based Networks operate efficiently, as the extent of the routing information maintained at each broker is a list of types or topics associated with each interested subscriber, reducing the complexity involved in matching incoming publications to stored subscriptions. When an incoming publication arrives at a broker, a simple look-up for the type/topic in an ordered list is performed, and when matched, is forwarded to all the subscribers associated with that type/topic. This is all conducted via the message header, which is marked, with one or more topic or types for each publication. In comparison both content and semantic-based networks see the incoming publications contents being evaluated against the brokers' stored subscription filters.

Content-based subscriptions are usually constructed from one or more filters in the form of a **Name-Type-Operator-Value (Attribute-Constraint)** and publications are constructed from one or more **Name-Value-Type (Attribute-Values)**. For each filter in each subscription the **Type** and **Value** must be compared, using its **Operator** against each **Type** and **Value** in each element of every incoming publication. If a match occurs the publication is forwarded towards the subscriber.

The time taken to search incoming publications against stored subscriptions depends on the particular set of subscriptions or publications. Importantly, with Type/Topic-based Networks the matching performance is predictable against the throughput of publications to subscriptions. For Content-based Networks performance is a function of the size (where the size = number of elements) in both subscriptions and publications and hence is less predictable. In Semantic-based Publish/Subscribe this problem is exacerbated as different semantic operators hold different associated costs. There is therefore a trade-off between the efficiency of Topic- and Content-based Networks like Scribe [10] and Siena [2] and the expressiveness of SBPS.

In aiming to reduce these costs one key optimisation of the routing information stored at each broker is subscription aggregation (covering), where similar subscriptions, which cover one another, are grouped together. Covering, in its simplest form, can be described as follows: Given two subscriptions **S1** and **S2** where each is made up of one or more filters **F1..Fn**, it can be said that **S1** covers **S2**, if all of the pubs that would match **S1** are a subset or equal to the set of

publications that match s_2 . Mühl et. al. [11] describe a benefit of this covering as being “a reduction in the number of entries in brokers’ routing tables.” If covering reduces the total number of routing entries in the broker, the matching and routing time involved in delivering publications to subscribers via those message brokers on which no covering has occurred is also reduced through a reduction in the number of subscriptions which must be matched against incoming publications.

In this thesis work is conducted on a particular class of SBPS, Knowledge-based Networks (KBN), a specific implementation of a KBN (KBNImpl) extended to form KBNCluster. KBNCluster focuses on incorporating the expressivity of SBPS with the efficiency of content-based routing and subscription aggregation through subscription clustering. The continuing research challenge in SBPS is where the increase in expressivity results in an additional increase in the overhead involved in determining matches between semantic subscriptions and publications. This research challenge has motivated the solution proposed in this thesis, aiming to reduce the average number of hops and thus brokers required to deliver messages, where the time taken to deliver a publication is directly related to the number of hops taken in its routing and thus overall load on the broker network. This is an alternative to trying to further optimise individual operator match performance [6]. In clustering this allows publishers and subscribers, with shared interests, to be arranged closer to one another in the broker network and thus the number of hops that a publication is required to traverse is reduced, therefore reducing overall load on the network.

More precisely when matching semantic terms in a publication to subscription, logical structures are used that make explicit a graph that expresses the ontology being referenced. Because this graph exists in definition, it can be analysed by its characteristics and the specific references made to it by publishers and subscribers in matches between publication and subscription.

It is assumed that shared interests will (as expressed by publications and subscriptions) involve terms nearer to each other in the semantic graph or ontology. Therefore clustering publishers and subscribers, using their semantic publications or subscriptions is predicted to reduce the number of hops a publication passes over in being routed to interested subscribers. This is in contrast to non-semantic CBNs, which operate only on the content (numbers, strings, sets etc of messages) and which do not have an explicit semantic structure that can be efficiently utilised in clustering. Publish/subscribe topologies generally allow these clients to connect to brokers in an unconstrained manner; meaning that the range of subscriber or publisher interests attached to any given broker cannot be predicted.

The approach to clustering, applied in this thesis, exploits the use of explicit semantic annotations in the publications and subscriptions not previously possible with non-semantic content-based messages. Once the explicit semantic makeup of a client’s publication or subscription is given, it

becomes possible to compare it to that of other clients publications or subscriptions, and thus create clusters of common interest, around message brokers. In support of clustering, the performance of the publish/subscribe network is predicted to be improved as follows:

1. The fewer hops (brokers) over which a publication message passes when being routed from publisher to relevant subscribers decreases both the time taken to deliver (based on per broker matching time and number of brokers involved in routing.)
2. Clustering reduces the semantic range of subscriptions that are received by brokers and the number of filter matches that must be made, on each broker. With clustering, a publication arriving at a broker, has a higher chance of matching a stored subscription, in that broker's routing table. The same routing table is also better optimised in terms of subscription covering, as subscribers are clustered according to the semantics of that subscription and thus the probability of subscription covering also increases.

In this thesis two types of clustering are defined, as discussed below:

1. **Static Clustering** instructs the brokers, publishers and subscribers of their relationship between one another pre deployment. However if this relationship changes, the statically embedded logic in every client must also change, usually with a full re-start. With the relationships between publishers, subscribers and brokers potentially changing rapidly, such a static approach is seen in [7] as being inefficient in terms of the ability of a network manager to manage efficiently the relationship between publishers, subscribers and clusters. In static clustering every publisher or subscriber must be manually instructed, by a network administrator, as to the broker they should attach to and it is thus clear to see that this does not scale as the number of clients increases.
2. **Managed Dynamic Clustering** allows publishers, subscribers and brokers to hold no statically embedded logic or knowledge of how to cluster. All logic is maintained in a management entity, which enforces the rules of clustering across the network, using an embedded management agent in each client and broker to enforce managed dynamic clustering. The key motivator for such an approach is that changes in managerial policies can be made and enforced without requiring any change on the part of the client.

To summarise: managed dynamic clustering happens automatically, as required, without having to re-start brokers. External intelligence is involved in optimising the decision process. The reasons behind taking a managed dynamic approach is that the best clustering configuration depends on the shared interests between clients, and therefore cannot be determined accurately prior to network configuration.

1.1 Research Question

This thesis examines whether a managed dynamic clustering approach improves the performance of Knowledge-based Networks, a sub-class of Semantic-based Publish/Subscribe.

1.2 Research Objectives

Three main research objectives have been drawn from the above research question:

Objective 1: Establish an approach for the formation, movement and re-clustering of semantic clusters in Knowledge Based Networks.

Objective 2: Establish the effect and overhead of implementing static and managed dynamic clustering in Knowledge Based Networks.

Objective 3: Apply Policy-based Network Management as an adaptable approach to the management of clustering.

In order to address *research objective 1*, a dynamic process for partitioning an ontology into a number of semantic clusters is presented. This partitioned ontology is then overlaid onto a broker network. This process directs into which cluster a subscriber or publisher should be placed. In addition, a misplaced subscriber can be identified and moved to a more suitable cluster.

In order to address *research objective 2*, it is shown that clustering decreases the average distance a message travels from publisher to subscriber. In addition to this, clustering is shown to reduce the overall load on individual brokers, and the broker network in general.

In order to address *research objective 3*, Policy-based Network Management (PBNM) is applied to control the process of clustering. This use of Policy-based Network Management is investigated in terms of the efficiency of management data collection, storage and policy execution.

1.3 Methodological Approach Taken

This research was conducted using an iterative investigative approach in evaluating the effect clustering has upon knowledge-based networks. The positive effect of clustering had been shown in topic-based networks. In content-based networks clustering became more difficult as there exists no agreed model of knowledge used to form messages. With semantics and knowledge-based networks a common model is present for use in clustering opening a new avenue of research.

However there was no starting point on which to build using a comparative approach. There was no system in which the effects of clustering, as shown in topic-based networks, had been applied in a knowledge-based environment. Having established that a comparative approach was not possible, this research took an approach based on evaluating incremental improvements in KBN performance through clustering. This was conducted using a two-pronged methodology evaluating system performance and network delivery metrics in parallel. These two metrics offer a mechanism for evaluating whether the introduction of clustering had either a positive or negative effect on the network. An un-clustered KBN Implementation was used as a benchmark to test the effect of clustering upon performance. This was conducted by comparing the results of the same experiments in both a clustered and un-clustered topology.

This thesis itself provides, for future researchers, a base-line implementation, evaluating performance gains through clustering which can be built upon by others, using a comparative approach, looking for incremental improvements. The comparative methodology that was not available in the early stages of this research is now available to others especially when combined with the future work section of this thesis.

1.4 Evaluation Findings

In this section the evaluation findings from this research are outlined and briefly discussed. The factors used in evaluating this work these are:

- Message delivery hop count, which refers to the number of brokers a publication passes through in being delivered to the subscriber as a notification. This is shown to related to the time taken to deliver a notification referring, to the period between when a publisher inserts the publication into the network, and the point at which a subscriber receives it.
- Subscription tree/set size refers to the number of subscriptions held by each broker at any given point in time.

An initial study was conducted into the effect that static clustering had upon a KBN deployment. This study concluded that clustering of publishers and subscribers around brokers of common interest increased the performance of KBNImpl in terms of subscription tree size and the number

of brokers involved in routing message from publisher to subscriber (Hop count). *A full evaluation of the approach taken to static clustering is included in the Evaluation Chapter 6, Section 6.2.* However in order to be able to dynamically cluster clients around brokers of interest, a method for extracting the semantic interests of publisher, subscriber, or broker is introduced. This returns the client's "Medoid" where this represents, in a single ontological value, the central ontological interest of the client. (A full definition of Medoid calculation is presented in the Design Chapter 4 Section 4.3) Once this Medoid has been calculated, clustering is achieved by placing publishers and subscribers around specific brokers that share, or have similar, Medoids to that of others. *A full evaluation of this can be found in Chapter 6, Section 6.5.3 entitled "Hop Count in Delivery."* It will be shown, **as a confirmatory finding**, that the greater the number of hops taken in routing a message, the greater the time taken for delivery. Therefore, with fewer brokers involved, the routing process is deemed to be more efficient. *A full evaluation of this can be found in Chapter 6, Section 6.3.3 entitled "Hop Count Experiments."*

Two types of node movement are introduced and evaluated in terms of efficiency:

1. Client based, where individual publisher or subscriber clients are moved from one broker to broker across the network..
2. Broker based, where a broker and all its directly attached clients are moved in unison from one position in the network topology to another.

It is shown that broker based movement requires less time and is more efficient than client based. *These movement methods have been evaluated and can be found in Chapter 6, Section 6.4.3 entitled "Mobility Costs."*

Collecting management information from brokers, publishers and subscribers, across the network, will be shown to be expensive, depending on the type of information being collected. The process of clustering has been designed, as discussed in the Design Chapter 4, so that this cost is reduced. *A full evaluation of this can be found in Chapter 6, Section 6.4.2 entitled "Data Collection Costs."*

Storing management information is evaluated in terms of the amount of memory used. In addition policy execution is evaluated in terms of the time taken to execute a number of policies against the set of previously stored management objects. *A full evaluation of this can be found in Chapter 6, Section 6.4.1 entitled "Management Data Storage & Policy Execution Costs."*

Finally a dynamic approach taken to clustering publishers and subscribers, around common brokers of interest, is evaluated in a set of experiments comparing the clustered implementation of KBNCluster to an un-clustered KBNImpl deployment. Clustered and un-clustered subscription tree sizes, hop counts taken to deliver messages and re-clustering message notification for clients deemed to be in an un-suitable cluster are compared side by side. *A full evaluation of this can be found in Chapter 6, Section 6.5 entitled "Dynamic Clustering Evaluation."*

1.5 Thesis Contribution

Much research has been conducted into clustering Topic-based Networks, [12] [10] [13] [14] [15], and is discussed in the State of the Art Chapter of this thesis. However the clustering of Content-based Networks has received less attention. Of the publish/subscribe architectures reviewed by Querzoni [16] in five operate as Topic-based Networks, whereas only two operate as Content-based Networks and none are semantic-based. The small number of studies into clustering content-based publish/subscribe is due to the difficulty associated with extracting and reasoning about the interests of publisher and subscriber messages without a formal semantic model. Although Querzoni's paper is only a single review of publish/subscribe architectures and their approach to clustering, his arguments are supported by the following **problem**: In topic-based publish/subscribe, clusters are easily formed around the taxonomy of topics used for publication and subscription. In content-based publish/subscribe, clustering becomes more difficult and involves subscription table comparison between brokers to calculate similarity between their subscribers. The **solution** however exists in Semantic-based Publish/Subscribe where an external model of knowledge, an ontology is used. This makes calculating semantic interests between clients an easier task, leading directly to the major contribution of this thesis, which is the ability to increase the scalability of semantic publish/subscribe networks through the introduction of clustering. The major **contribution** of this thesis is therefore formed from the ability to calculate the semantic centre of a publisher, subscriber or broker and thus allows common interest groups, or clusters, to be formed. The **impact** of this work compares an approach for clustering clients in semantic publish/subscribe systems with the ease at which it is achieved in topic-based publish/subscribe thus bypassing the problems of clustering Content-based Networks.

An additional minor contribution of this thesis is the approach used in managing the clustering of KBNs. The **problem** relates to managing a process of dynamic clustering across a network of brokers, publishers and subscribers. The **solution** is to use an approach that enforces clustering using defined policy rules and a policy engine with data **that is collected and filtered by a semantic publish/subscribe message broker**. Others have used Policy-based Network Management (PBNM) in [17] to determine what tasks to perform post publication-subscription-matching, and Role-based Access Control (RBAC) [18] has been used to restrict access to content across the publish/subscribe paradigm. However the **novel contribution** of this thesis allows for management agents embedded on each node to be instructed by a policy server as to how to **cluster clients**, allowing for changes in managerial goals to be enforced in changes in policy, as opposed to changes in the client's code base. The **impact** of this work relates to the approach taken in applying an approach to clustering and filtering management data, sourced from across a collection of managed nodes, and using semantic publication and subscription delivery via a KBN broker network for management message delivery.

1.6 Selected publications

Seven peer-reviewed publications form the basis of this thesis. The full set of contributed publications is included in the Conclusion Chapter 7, Section 7.2. In this section a number of these publications are outlined and a brief description is provided as to how they add to the State of the Art in publish/subscribe clustering or publish/subscribe research in general. These publications were chosen as they represent the three main concepts of this thesis: Knowledge-based Networking, managed clustering using policy-based networking, and an approach to semantic clustering.

- John Keeney, Dominic Jones, Song Guo, David Lewis, and D. O'Sullivan, "**Knowledge-based Networking**": book chapter, published in the "Handbook of Research on Advanced Distributed Event-Based Systems, Publish/Subscribe and Message Filtering Technologies. "IGI Global (Editor(s): Annika Hinze and Alejandro Buchmann) 2009.
 - This peer reviewed book chapter introduces and presents a complete overview of the Knowledge-based Network, an implementation of which is used in this thesis. Work from this thesis contributed to the related work, motivational case studies and discussion/future work sections of this chapter.
- Dominic Jones, John Keeney, David Lewis, and D. O'Sullivan, "**Policy-based Management of Semantic Clustering**": conference paper, presented at the second International Conference on Distributed Event-Based Systems (DEBS 2008), Rome, Italy, July 2008.
 - This paper presented the policy-based approach used within this thesis for controlling the clustering of publishers, subscribers and brokers. The application of using policy in publish/subscribe clustering is a novel contribution of this research. In this position paper the work in this thesis outlined the approach evaluated in the rest of this thesis.
- John Keeney, Dominic Jones, Dominik Roblek, David Lewis, and D. O'Sullivan, "**Knowledge-based Semantic Clustering**": conference paper, presented at the twenty third annual ACM Symposium on Applied Computing (SAC 2008), Fortaleza, Brazil, Mar 16-20 2008. (Included in Appendix A)
 - This conference paper presented and evaluated a static approach to clustering, where each client was manually configured with a pre-defined cluster. This work supported the arguments on the benefit to KBN clustering motivating the research presented in this thesis.

1.7 Thesis Outline

Chapter 2, Background: This chapter provides the reader with the necessary background knowledge for the thesis. By the end of this chapter the reader will have acquired an understanding of publish/subscribe networks, ontologies and policy-based network management.

Chapter 3, State of the Art: This chapter places the work presented in this thesis within the context of existing research in both publish/subscribe networks and the clustering of these networks. By the end of this chapter the reader should be able to compare the research presented in this thesis against research completed by others, and see where this research contributes to the State of the Art.

Chapter 4, Design: This chapter presents the reader with a description of the design decisions made in the process of clustering of Knowledge-based Networks. By the end of this chapter the reader should have an understanding of how each part of the system connects together.

Chapter 5, Implementation: The reader will have gained an understanding of the various components of the evaluated system from the design chapter. This chapter introduces the various technologies and technical approaches taken to implement a proof of concept of the design, KBNCluster.

Chapter 6, Evaluation: The evaluation chapter presents the findings of the research conducted as part of this thesis. Each of these findings is supported in data. By the end of this chapter the reader will be able to identify the benefits of clustering KBNs and the evaluated efficiency of the management system.

Chapter 7, Future Work and Conclusions: This chapter presents the future direction of this research. In addition, it presents an overview of the research question and objectives backed by supporting data. Finally the benefits to the scientific community of this work are clearly outlined and discussed.

Appendixes: **A:** Full copy of "Knowledge-based Semantic Clustering" [7]. **B:** Results of the clustering algorithm used in this thesis applied to five separate ontologies. **C:** All policies designed as part of this thesis. **D:** In DVD format: a copy of this thesis in PDF format, all evaluation data sets, publications and subscriptions used in experimentation, all ontologies (in *.owl format) results of the clustering algorithm applied to the ontologies, all policies used in as well as copies of all papers published from the work presented in this thesis.

2 BACKGROUND

This chapter presents the technology and background knowledge important in fully understanding this thesis. In this chapter, ontologies, semantics, semantic reasoning, publish/subscribe middleware and Policy-based Network Management (PBNM) each are introduced in turn.

2.1 Key Terms

This first section describes for clarity some of the key terms in the context of this thesis, not specifically to the field of publish/subscribe.

Subscribers register subscriptions with a broker. Each subscription consists of one or more filters, where each filter is constructed from a *Name*, *Type*, *Operator*, and *Value* in the form of an *Attribute-Constraint*. **Publishers** send publications into a broker across the network constructed from a *Name*, *Type*, *Value* in the form of an *Attribute-Value*. These publications are matched, by message brokers, to stored subscriptions and routed across the broker network towards subscribers as notifications, where matches occur.

Brokers route messages received from a publisher to a subscriber. Subscriptions are stored in a broker's subscription set or tree. When a broker receives a publication, it checks its subscription set and forwards matches towards the subscriber that was the source of the subscription.

Hop Count relates to the number of brokers over which a publication passes as it travels from its source to a destination. It is initialised at zero, and the first broker to receive a publication increments it by one. This continues until the publication turns into a notification, when it is delivered to a subscriber with an attached final count.

An **Un-clustered Topology** is a broker topology where both subscribers and publishers are unconstrained in the choice as to where they publish or subscribe to across the broker network. There is no coordination as to where, in the overlay, from a logical point of view, a publisher or subscriber should connect. For the purposes of this thesis, a **Clustered Topology** utilises a managed process to attach both publishers and subscribers to an appropriate broker before they publish or subscribe, thereby clustering the clients around their interests. When a Publisher or Subscriber requests attachment, its semantic interests are used to calculate a suggested cluster within the network.

The process used to control clustering takes a Policy-based Management [19] approach. The Rule Driven **Policy Server** plays a central role in the management of the clustering process. Management policies are designed and implemented by a systems administrator. These policies and the methods these policies utilise subsequently allow for a clustering process to be implemented across the managed collection of nodes by a centralised policy server.

2.2 Semantics

2.2.1 RDF and Ontologies

In this section ontologies are introduced. However before this, the Resource Description Framework (RDF) is briefly discussed. Baader et. al. [20] define RDF as “a language for representing information about resources in the World Wide Web” written so that machines can parse it, and humans read it. In RDF, a resource can be either a *Subject*, *Object*, or *Predicate*, each with a unique Uniform Resource Identifier (URI). Stringing together a subject and an object with a predicate creates an RDF triple. Importantly objects and predicates can also form parts of other triples, and these can be extended into one another so that they form a knowledge base. By way of illustration, a simplistic example of an RDF is shown below:

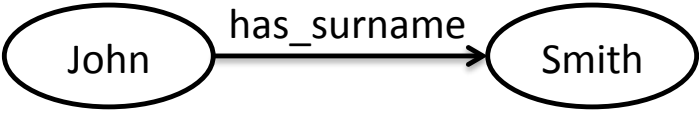
<p><i>Subject:</i> <i>John</i></p> <p><i>Predicate:</i> <i>has_surname</i></p> <p><i>Object:</i> <i>Smith</i></p>	
--	--

Table 1: Example RDF Triple

Moving on from RDF, Web Ontology Language (OWL) ontologies are seen as more detailed and formal representations of knowledge, formed from classes, instances/individuals and properties. The remainder of this section will introduce the main characteristics of a generic ontological model in the course of building a simplistic example ontology.

1) **Classes:** represent collections of individuals that share common characteristics. In the example ontology built through this chapter a single root class is created, underneath *Owl:Thing*, where *Owl:Thing* represents all knowledge in the modelled domain.

The newly created class sub-class of *Owl:Thing* will be named *Animals* and have additional sub-classes of *Cats* and *Dogs*. Classes named *Burmese* are created under *Cat* and *GermanShepherd* and *JackRussell* under *Dog*. Using this approach sub- and super-classes can be extended into a taxonomy as required. Equivalence and disjoint relationships can be encoded into class relationships, such that *Dog* and the class *Canis lupus familiaris* can be defined as being equivalent and *Cat* defined as being disjoint from *Dog*.

2) **Instances/Individuals:** are occurrences of entities and assigned to specific classes. By extending the class structure created in the previous step, *Dixie* is associated to the class, *Burmese*, *Tillie* and *Ray* to the class *German Shepherd* and *Gus* to the class *Jack Russell*. This process populates the previously created class structure with instances of each class. The next step assigns some properties to the sample ontology.

3) **Property Types:** The two main properties used in OWL ontologies are either object- or data-type properties. The W3C defines object-type properties, in [21], as a directional relationship between “*an individual(s) and another individual(s)*” and data-type properties as “*individual(s) to datatype(s)*” where data types equal strings, integers, Booleans, dates etc. When dealing with object properties, the domain and range can be set. For data-type properties an instance is directionally related to typed data values expressed in XML schema.

Expanding upon the example ontology a property is created where the **Domain** is set to **Dog** and the range is set to **Cat**. Once this property has been assigned a domain and range and subsequently when an instance of the **Dog** class is selected it is possible to assign the property **chases** to the selected **Dog**. For example **Tillie**, a **German Shepherd** chases **Dixie** a **Burmese**.

An example of data-type properties is **hasAge** which has a domain of **Animal** and its range being set to **Integer**. This allows any instances of any sub-class, below **Animal**, to be assigned an integer value. Shown in Figure 1 is the example ontology formed in this section.

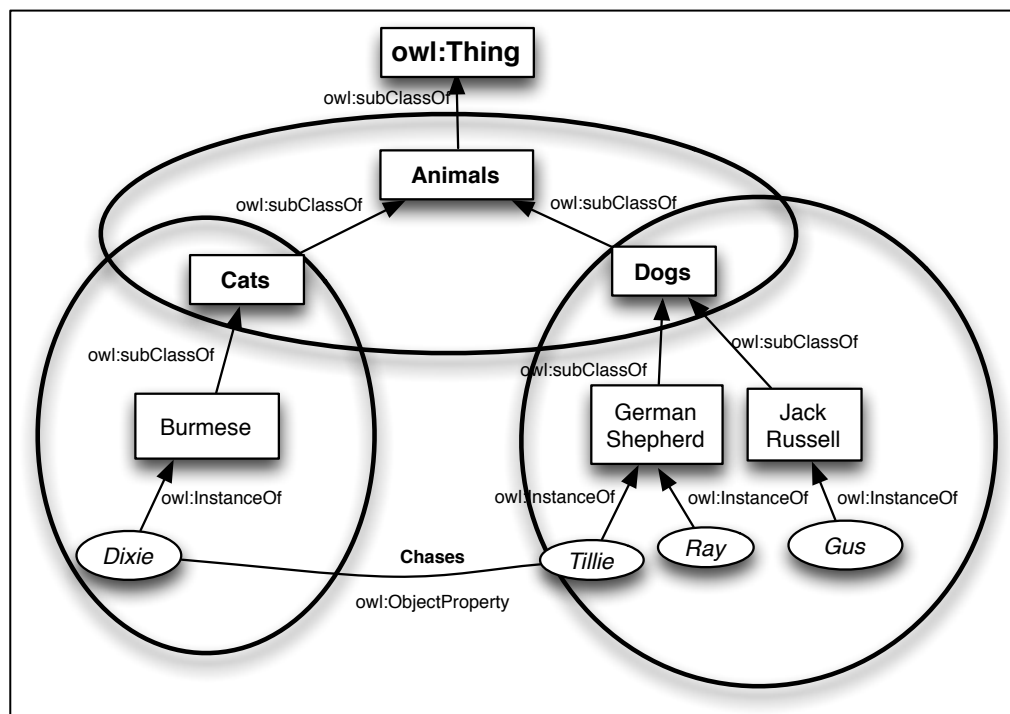


Figure 1: Example Ontology

One limitation of the ontological representation of knowledge is in the ability of a machine to understand the meaning of that knowledge. Whether in OWL, RDF, or plain text, a machine can only understand the relationships among concepts based on the path between them, not by the semantic meaning of the concept’s label. At present it is difficult for a machine to understand meaning but it is possible to identify relationships between concepts in a particular knowledge model. Taking an object-property **Father is parent of Child** a machine could interpret

through reasoning the link between the concept of *Father* and of *Child*, but not of the meaning of the wording of the property, *is_parent*.

If ontologies are seen as multi-dimensional graphs of knowledge, where relationships are represented as formally defined properties, super- and sub-class, assignments of instances and properties it is thus possible to refer knowledge from these relationships whereas understanding the concepts is more difficult to achieve. There are two main methods for inferring knowledge, as outlined in [22], the T-Box or “Taxonomy Box” and A-Box or the “Assertion Box.” Combined, they apply a classification of concepts and the relationships between concepts individuals and properties, which provides a query-able reasoned taxonomy of a knowledge base.

2.3 Publish/Subscribe Middleware

Having discussed ontological semantics this section looks at the different types of publish/subscribe middleware responsible for delivering event messages from publishers to subscribers, whose interests are expressed as a subscription. This approach can be viewed in contrast to systems such as (RSS) Really Simple Syndication [23] where a pull-based delivery mechanism is used. In RSS a client is required to query for new content from a defined collection point, as it appears, whereas the push-based publish/subscribe paradigm sees content pushed towards a subscriber, with no action required on the part of the subscriber.

Distributed publish/subscribe offers a de-coupled method of communication in which messages are routed from a publisher towards interested subscribers via a broker without any direct relationship existing between both. Agreement is only required in terms of the message format and the broker mechanism in use, often achieved through a boot-strap service. The remainder of this section looks, in detail, at three types of publish/subscribe middleware, topic-, content- and knowledge-based.

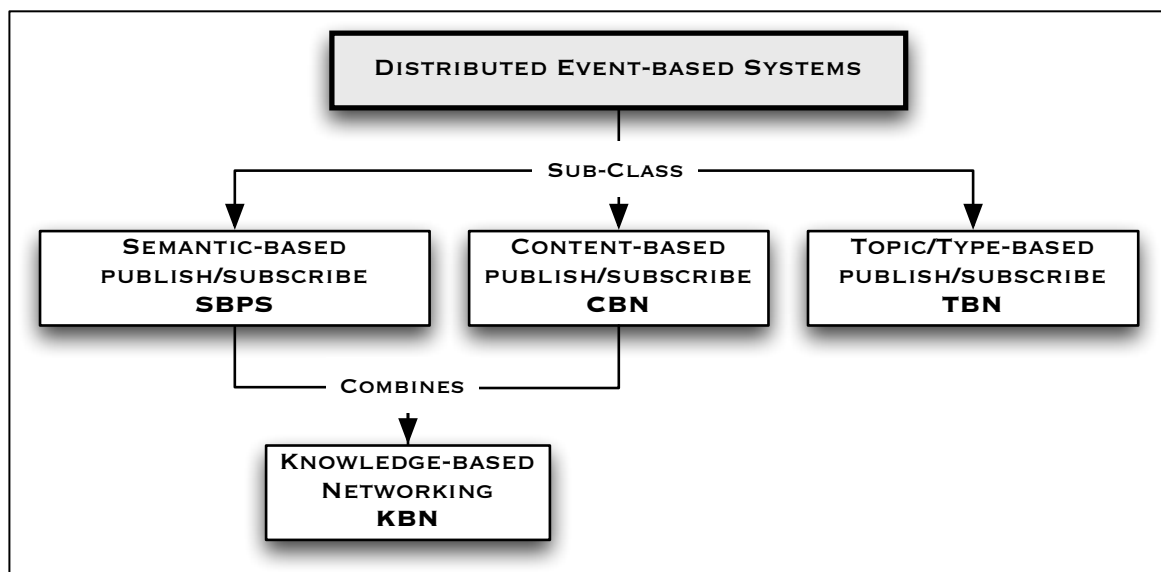


Figure 2: Classification of Publish/Subscribe Systems

Shown in Figure 2 are sub-classes of Distributed Event-based Systems (DEBS) [24] discussed in the last three sections of this chapter. In this thesis the prototype system developed upon is an implementation of Knowledge-based Networking [9] which itself extends upon the Siena [2] Content-based Network (CBN). The State of the Art Chapter 3, of this thesis, extends the above diagram with the various implementations of the four classifications presented above, introducing topic-, content- and Knowledge-based Networks in turn.

2.3.1 Topic-based Publish / Subscribe:

A highly cost-efficient and widely used publish/subscribe system is the topic-, type- or channel-based, network where clients subscribe and publish to particular topics or channels of information, each a specific reference to a stream of information. An example of the concept of subscriptions in such a system could include topics such as *Wimbledon* as a sub-topic of *Tennis* or *World_Cup_Final* as a sub-topic of *Football*. Subscribers are delivered all messages published to the specific topic-channel they have previously subscribed to. TIBCO [1] and Scribe [10] are examples of popular Topic-based Networks.

One requirement of topic-based publish/subscribe is that there must exist an agreement on the topic set, pre-defined by an administrator. De-coupling of clients is a key characteristic of publish/subscribe, yet the requirement for agreement upon a topic set reduces this de-coupling and requires a centralised service to advertise available topics. However, this is offset by the efficiency of routing messages to pre-defined topic channels, therefore in Topic-based Networks the only part of the message available to the event-service used in message matching is the topic identifier itself, included in the message header.

2.3.2 Content-based Publish / Subscribe

Content-based Networks (CBN), such as Siena [2], Hermes [25], Gryphon [26] and Elvin [27], allow subscriptions to be matched against the contents of publications. In CBNs publications are usually formed from a number of attributes composed of a *name*, *type* and *value*, whereas subscriptions are a conjunction of one or more *name*, *type*, *operator* and *value* filter(s). If a subscription is satisfied, from the contents of a publication, the publication is forwarded to the subscriber as a notification.

Type	Name	Operator	Value
<i>String</i>	<i>User_Name</i>	=	<i>jonesdh</i>

Table 2: Example Filter Constraint

Shown in Table 2 is a content-based subscription, constructed using a single filter in the form of a *Type*, *Name* an *Operator* and a *Value*. Each of the filters in a given subscription must be matched to a candidate publication, if it is to be matched as a notification. However there may be more attributes in the publication than specified in a matching the subscription, but every part of the subscription must be satisfied by one or more parts of the publication.

Type	Name	Value
<i>String</i>	<i>User_Name</i>	<i>jonesdh</i>
<i>int</i>	<i>MailBoxSize</i>	<i>500</i>
<i>int</i>	<i>ExtNumber</i>	<i>6099</i>

Table 3: Example Publication

The example publication shown above in Table 3 would be matched to the subscription shown in Table 2 as all of the subscription filters are satisfied. In contrast to this, shown in Table 4 is a **non-matching subscription**, as the *MailBoxSize* attribute does not match any of the attributes in the example publication, and therefore will not be delivered to the subscriber.

Type	Name	Operator	Value
<i>String</i>	<i>User_Name</i>	=	<i>jonesdh</i>
<i>int</i>	<i>MailBoxSize</i>	>	<i>590</i>

Table 4: Example Subscription

2.3.3 Knowledge-based Networks in comparison to CBNs

Knowledge-based Networks (KBNs) [9] are a classification of semantic-publish/subscribe middleware, allowing the semantics of the contents of publications and subscriptions to have an effect in message matching.

The semantic content in subscriptions and publications are based on a shared semantic web OWL ontology [3], using additional ontological types and operators. A key identifier of Knowledge-based Networks is that semantic types and operators are often integrated seamlessly with the existing (non-semantic) types and operators of CBNs, the KBN implementation used in this thesis offers an extended version of a CBN (Siena [2]) in which both semantic and non-semantic message attributes operate in unison, across the same network of brokers being matched either syntactically (available in both CBN + KBN) or semantically (only available in KBN).

One problem that exists with Knowledge-based Networks is in the requirement for pre-agreement on the common model of knowledge, or Ontology, used between clients and brokers. This pre-agreement is a drawback as it reduces the de-coupled nature of publish/subscribe however it also provides an increases level of expressivity in messaging. KBNCluster does not address the issue of pre-agreement between ontological models nor does it require clients to communicate in a pre-operational agreement phase. However the work of Guo [4] looks in detail at the use of ontological mappings between different ontologies so to mitigate the problem of pre-agreement on ontological models as present in Knowledge-based Networks.

The data in Table 5 introduces Knowledge-based Networks by comparing them against both Topic- and Content-based Networks in terms of delivery, subscription and publication method, available operators, pre-operation agreements as well as providing examples of each, from current literature.

	<i>Generic Topic-Based</i>	<i>Siena [28] CBN</i>	<i>KBNImpl [8] KBN</i>
Delivery Method:	Each topic channel has all messages delivered to all clients, who have previously expressed interest in the channels content.	Messages are delivered by brokers based on non-semantic subscriptions constructed from a Name, Type, Operator and Value .	Messages are delivered by brokers based on non- and semantic-subscriptions constructed from a Name, Type, Operator and Value .
Subscription Method:	Subscriptions to topic channels are seen as expressions of interests in all content to be published to that channel.	Name-Type-Operator-Value (Attribute-Constraints) using syntactic subscriptions formed from non-semantic operators and types.	Name-Type-Operator-Value (Attribute-Constraints) syntactic and semantic subscriptions using both non-semantic and semantic operators and types
Publication Method:	Publication pushed into specific channel and delivered to all subscribers of that channel.	Non-Semantic publications Name-Value-Type (Attribute-Values) matched at each broker against stored subscriptions, delivered as notifications to all interested subscribers.	Non-Semantic and Semantic publications Name-Value-Type (Attribute-Values) matched at each broker and delivered as notifications to all interested subscribers.
Operators available:	Limited set, based around exact subscriptions to previously agreed topic taxonomy, examples include: Sub-, Super-, exact or equivalent operators.	Including, but not exclusive of: Equals =, Not Equals !=, Less Than < and <= Greater Than > and >=, Prefix (String starts with) >*, Suffix (String ends with) *<, Substring * , Equal Bag #=, Sub Bag #< , Super Bag #>, (Logical OR , Logical AND &&, Logical NOT ! Varies for different implementations.)	Including, but not exclusive of: Equals =, Not Equals !=, Less Than < and <= Greater Than > and >=, Prefix (String starts with) >*, Suffix (String ends with) *<, Substring * , Equal Bag #=, Sub Bag #< , Super Bag #>, Logical OR , Logical AND &&, Logical NOT ! Equivalent @~ , Not Equivalent @!~ , Sub Class @> , Super Class @< , ISA @= , IS_NOT_A @!= , Ont Property @* ,
Operational agreements required:	Agreement of topic taxonomy by publishers, subscribers and brokers.	Requires agreement on names and types between publishers, subscribers and brokers.	Requires agreement on an ontology used by publishers, subscribers and brokers and name and types used.
General Examples:	TIBCO [1] and Scribe [10].	Siena [2], Hermes [25], Gryphon [26] and Elvin [27].	KBN [9], OPS [29], and S-ToPSS [30]

Table 5: Classification of Topic, Content and Knowledge-based Network

Shown in Table 6 are the range of operators and applicable types available in the Siena CBN [2]. The combination of these operators and types, with a value, allows *Attribute-Constraint-filters* to be formed from one or more *Name, Type, Operator and Value*.

Operators	Symbol	Applicable Types
<i>Equals</i>	=	<i>String, Byte Array, Integer, Long Integer, Double, Float, Boolean.</i>
<i>Not Equals</i>	!=	<i>String, Byte Array, Integer, Long Integer, Double, Float, Boolean.</i>
<i>Less Than</i>	<	<i>Integer, Long Integer, Double, Float</i>
<i>Less Than Equal To</i>	<=	<i>Integer, Long Integer, Double, Float</i>
<i>Greater Than</i>	>	<i>Integer, Long Integer, Double, Float</i>
<i>Greater Than Equal To</i>	>=	<i>Integer, Long Integer, Double, Float</i>
<i>Prefix (starts with)</i>	>*	<i>String</i>
<i>Suffix (ends with)</i>	*<	<i>String</i>
<i>Substring</i>	*	<i>String</i>

Table 6: Siena CBN Operators, Types and Symbols

Shown in Table 7 are the operators and applicable types available in the KBN implementation used, extending the CBNs operators and types, shown in Table 6, to include semantics.

Operators	Symbols	Applicable Types
<i>CBN Operators.</i>	<i>See Table 6.</i>	<i>See Table 6.</i>
<i>Equivalent</i>	@~	<i>Class or Property</i>
<i>Not Equivalent</i>	@!~	<i>Class or Property</i>
<i>Sub Class</i>	@>	<i>Of a named Class</i>
<i>Super Class</i>	@<	<i>Of a named Class</i>
<i>ISA</i>	@=	<i>Individual against Class</i>
<i>IS NOT A</i>	@!=	<i>Individual against Class</i>
<i>ONT_PROP</i>	@*	<i>Individual, Property, Individual</i>

Table 7: KBNImpl KBN Operators, Types and Symbols

Roblek introduces, in [31], shown in Table 8, Bag operators and types. A Bag can contain any valid KBNImpl (Table 7) type, including other Bags and their applicable operators in the form of subscriptions or as publications. Examples of matching a publication (Bag A) against a subscription (Bag B) are as follows: *Equal Bag*: all elements in Bag A must be in Bag B. *Sub Bag*: Some elements in Bag A must be in Bag B. *Super Bag*: Some elements in Bag B must be in Bag A. Each of these combinations is represented below, in Table 8, Bag operators and types.

Operator	Symbol	Applicable Types
<i>Equal Bag</i>	#=	<i>See Table 6 and Table 7.</i>
<i>Sub Bag</i>	#<	<i>See Table 6 and Table 7.</i>
<i>Super Bag</i>	#>	<i>See Table 6 and Table 7.</i>

Table 8: Bag Operators, Types and Symbols

In this section three forms of publish/subscribe middleware have been introduced, topic-, content- and knowledge-based, where the latter is identified as a sub-class of Semantic-based Publish/Subscribe. These three classes of publish/subscribe middleware are compared to one another in tabular form before the operators, symbols, and applicable types for both content- and Knowledge-based Networks have been introduced.

2.4 Policy-based Network Management (PBNM)

The Design Chapter 4, Section 4.5, of this thesis outlines a number of dynamic management methodologies used in managing the clustering of clients who require a dynamic management approach. In this thesis the placement of clients is dynamic in that it is not based not on any pre-determined metrics, but on the specific semantic interests of the client at a given point in time, acquired during operation.

PBNM is seen as an appropriate choice for encoding enforceable management requirements as it enables for dynamic and rapidly changing rules to be integrated directly into the operation of the management process without any re-compilation or re-deployment of the management system. PBNM is defined by Boutaba et. al. as supporting this view in [19], where it is seen as “separating the rules governing the behaviour of a system from its functionality.” It is this level of separation between the policy system used for achieving clustering and the broker network itself that makes a PBNM approach highly suitable for this thesis.

PBNM simplifies the management of this thesis by declaring operating rules, which deal with situations that are likely to occur. Informally, policy rules can be regarded as a declarative instruction or authority for a manager to execute actions on a managed target to achieve an objective or execute a change. Supported by this premise, PBNM is used within this research to provide a set of external controls, which, in certain configurations, allow management level decisions to be filtered down from the networks management into the operational characteristics of a clustered KBN.

The goal behind using PBNM in the clustering of brokers and clients is to control the behaviour of brokers and clients by employing well-defined policy rules, so that an administrator can manage the network as an entity in-itself. PBNM allows this to be realized through its ability to implement changes across the network as a whole, in comparison to managing individual network entities and actions. PBNM often uses the principle of an *event-condition-action* loop, where the *event* is an occurrence of a trigger phenomenon within the system. The *condition* is defined by the policy manager as an arbitrary set of additional conditions that might refer to the event, the system, or other observable context. Finally the *action* represents the task to perform if the event received matches the conditions required for an action to be fired. The policy loop can be formalised as *IF [condition] THEN [action]*, an example being presented by Martin et. al. in [32]: *If [network resources are low] THEN [limit WWW access]*. Such an approach is used in designing the policies and enforcing the clustering, evaluated as part of this thesis.

2.5 Conclusion

In this Background chapter, an initial introduction to semantics has been outlined showing how RDF triples [20], consisting of a *Subject, Object and Predicate*, can be used to represent semantic data. These triples have been extended and discussed in relation to the three main types of OWL ontologies. A sample ontology has been built in terms of classes, individuals/instances and properties. Classes have been defined as representing collections of individuals, and individuals as representing instances of classes where object properties may be used to relate one individual to another. Data-type properties, in RDF, have been discussed as resources that relate to some value (which may be another resource.) However, in OWL object properties relate individual to individual, and data-type properties individuals. An OWL reasoner can follow these properties to inter-logical relationships between individuals and data based on a given ontology.

Various classes of publish/subscribe middleware have been introduced, including Topic-based, such as Scribe [10]; Content-based (CBN), such as Siena [2]; and Knowledge-based Networking [9] as a sub-class of Semantic-based Publish/Subscribe (SBPS). For each of these networks, the various types of operators and the various types that they apply to have been documented. It has been shown how topic structures are subscribed to and how content-based *Attribute-Constraint* filters are formed around *Attribute-Values*.

Finally Policy-based Network Management [19], [32] has been introduced and established in terms of various parts of a generic policy loop, in the form of *event-condition-action*. The next chapter looks at current State of the Art research in the field of publish/subscribe networks, their clustering, and approaches to managing clustering

3 STATE OF THE ART

In the previous chapter the technology and background used in this thesis have been introduced. In this chapter the current State of the Art in the field of publish/subscribe is introduced, concluding with design ideas, drawn from across the State of the Art. The first section of this State of the Art Chapter looks at other implementations of Content-based Networks. Following on from this a review of Semantic-based Publish/Subscribe (SBPS) is conducted, both sections concluding with the advantages / disadvantages of each technology. A detailed review of the application of clustering in both topic-, content- and Semantic-based Publish/Subscribe is conducted, before this chapter concludes with design ideas, applied in KBNCluster, sourced from the current State of the Art.

3.1 Content-based Networks

As has been discussed, Content-based Networks (CBN), such as Siena [2], Hermes [25], Gryphon [26] and Elvin [27] are formed around the necessity to match a varying subscriber base to a wide range of publishers. This “de-coupling” of the parties involved in the communication process allows for message routing, upon message brokers, based on subscribers *interested* in a particular message, rather than the flooding of the network in the search for all interested parties.

Subscribers register interests in the form of subscription filters, which are matched against incoming publications at a broker. The routing process allows for a message inserted into a broker network to be routed across the network based on positive matches to stored subscriptions (made up from a number of filters) until every client, interested in the message, is satisfied. The filters, which match these publications, are constructed using filtering constraints applied to the contents of the publications. Mühl et. al. [24] describes this content-based matching as a set of “filters [which] are evaluated against the whole contents of notifications.” The range of these operators and types determines the type of pub/sub network in which the message is being sent and these operators have been introduced and discussed in the Background Chapter 2, Section 2.3.

This section overviews four well known Content-based Networks, Siena [2], Hermes [25], Gryphon [26] and Elvin [27], concluding with the advantages and disadvantages of the various CBN architectures discussed.

3.1.1 Siena

Central to the design of Siena [2] is the observation that: “in practice, many parties are interested in similar events” central to the approach taken in clustering. This is put into practice when a subscription is received by a broker, as this subscription is only forwarded to a peer broker node if “it defines newly selectable notifications that are not in the set of selectable notifications defined by any previously propagated subscriptions.” Using this approach, the Siena broker network

removes some of the load placed on the processing of publications by only forwarding the subscription up the hierarchical chain of brokers, making it a concern of other brokers, if the publication matches previously forwarded subscriptions.

The relationship between advertisements (not used in this thesis) and publications in Siena are more complex. Siena is an event notification service, where publications are the events, compared to the set of subscriptions stored in each broker. Changes in subscribers' interests are represented by a change in the set of stored subscriptions they hold at a broker. Advertisements allow the publisher to "inform the event notification service about the notifications that will generate objects of interest," allowing brokers a glimpse of the content that may become available from a particular node in the future. Advertisements are non-binding; messages are delivered even if the advertisement is different from that of the subsequent publication.

Finally Siena identifies a single match of a subscription and a publication as a pattern. One or more filters form a subscription and a publication matching a subscription is termed a notification. This form can be further extended into a sequence of multiple filters, in a pattern, where each filter must be matched in order of definition in order for that complete pattern-subscription to be matched.

3.1.2 Hermes

As with other distributed event-based systems, Hermes [25] uses the terms event-clients and event-brokers, where event-clients are publishers or subscribers and event-brokers route messages between the two. However Hermes introduces the concept of "rendezvous" nodes, known to both publishers and subscribers placing each around a common shared broker of interest, where message delivery occurs or will occur. In addition to this, Hermes allows for subscription to type-based events. For each event type, a rendezvous node exists in the network, and is replicated somewhere else within the network. This assures that no single rendezvous node becomes a "single-point of failure," but also allows clients to connect at particular nodes that deal with particular event types.

Publications are delivered to subscribers in Hermes, as follows: publishers establish rendezvous (\mathcal{R}) nodes for each of the types on which they will publish. They then send advertisements across message brokers to these \mathcal{R} nodes. Subscribers similarly subscribe across broker paths, from their destination to the \mathcal{R} nodes relevant to their subscription type. As these subscriptions route across the collection of brokers, each individual broker stores the subscription locally, the path travelled terminating at the relevant \mathcal{R} node. Once the subscriptions have been processed from subscriber to \mathcal{R} nodes, and the path stored, brokers route publications back across these paths on behalf of publishers. As publications pass across brokers (towards \mathcal{R} nodes), brokers check to see if routing publications match any stored subscriptions. If a match does occur, the broker forwards the

publication, towards the subscriber, using the reverse route of the previously stored subscription. Using this approach Hermes attempts to avoid bottlenecks occurring if subscriptions route to an R node and only then route back to subscribers so that brokers take some of the load of the R node as soon as they can. A layer below the API allows for a mechanism to filter across the event type values, as opposed to just the event-type, much like the Siena filtering mechanism.

3.1.3 Gryphon

The authors of Gryphon [26] present three approaches for solving the multicast (publisher to subscriber) problem, initially using a match-first approach, secondly using a flooding approach, and thirdly presenting a hybrid approach. When match-first message matching is utilised, a list of matches is generated when a publication is received. Once this list is accumulated the publication is forwarded, as a notification, to each of the generated matches (subscribers). Flooding, as in the name suggests, pushes the publication to all nodes, regardless of whether a match may occur or not. The Gryphon authors outline and evaluate a third, more efficient, hybrid approach to the problem of message delivery, termed “link matching.” Central to the concept of link matching is the principle that each broker can calculate the route, from themselves to a subscriber, with respect to the brokers outgoing links when the broker receives a publication.

Gryphon then uses a Parallel Search Tree (PST) approach in an initial prototype which is then extended in [26] using a neighbour-first approach. In this, each subscription corresponds to a path from the root node of the PST to the corresponding subscriber. By following the tree from the root node down each of the possible attribute paths, the distance between each incoming publication and interested subscriber can be calculated, the shortest path being the one used in message delivery.

3.1.4 Elvin

In [27] the Elvin3 protocol is discussed in operation, and extended theoretically into Elvin 4. In Elvin3 clients are represented as producers (publishers), consumers (subscriber) and servers (router/broker). An important note with regard to Elvin3 is that the message delivery is conducted over a single message server, solely responsible for the collection of subscriptions and delivery of publications to interested subscribers. In their discussion regarding Elvin4 the authors discuss the problems involved in scaling such an event-based system, where a single server processes all load, whilst keeping intact the message-matching algorithm. In this discussion the authors conclude that: “The remaining research challenge is to address scalability to wide area networks, and to provide an internet-scaled Elvin [publish/subscribe] service.” This is important with regard to the design of publish/subscribe systems and specifically when dealing with a growing set of stored subscriptions and incoming publications where reducing the cost involved in matching each publication to each subscription, with such large distributed subscription table is an aim of design.

3.1.5 Analysis

3.1.5.1 Advantages of CBNs

Content-based Networks offer many advantages over their predecessor; Topic-based Networks. The use of Content-based Networks provides a more expressive and flexible subscription matching mechanism. This increased flexibility is achieved through the use of **Attribute-Constraint** filters that allow subscription filters to be compared to the separate constituent parts of **Attribute-Values** making up a publication. If one or more of the publication's attributes match all of the constituent parts of the subscription, then the message is forwarded to the subscriber as a notification. Content-based Networks allow for a range of operators to be added to the combination of **Attribute-Constraints**, present in subscription filters, such as; greater than, less than, equals, does not equal etc. This combination of attributes, constraints and filters, specified by the users, increases greatly the expressiveness of the subscription mechanism and thus the expressiveness of the system.

In Content-based Networks there is no requirement that the set of users neither join together in agreeing what subjects, sub-subjects and super-subjects should be formed into a topic taxonomy, nor use a pre-defined taxonomy only required is that they agree on naming conventions between attributes. Users publish, regardless of subscribers' interests, and the same is true for subscribers where content is delivered based on a match between subscriptions filters and publications, routed to one another from across the network.

3.1.5.2 Disadvantages of CBNs

There are however some drawbacks to Content-based Networks not apparent in Topic-based Networks. These drawbacks are primarily associated with the increased cost of matching stored subscriptions to incoming publications, where the complete contents of each publication needs to be compared to every subscription held by a broker, regardless of whether any match may be made elsewhere within the subscription tree. This results in an increase in operational costs when searching for notification matches, and subsequently increases if the number of **Attribute-Constraint** pairs in each subscription increases. However some pre-agreement is still required with Content-based Networks relating to the naming of attributes. Unless the naming of attributes is in synergy then publications will not match subscriptions, regardless of the content. The balance between the level of specification and the agreement of naming attributes is one that is constantly re-assessed. In summary, Content-based Networks are shown to be extremely expressive, but still require pre-agreement upon the naming of attributes and their pre-deployment.

3.2 Semantic-based Publish/Subscribe (SBPS)

Semantic-based Publish/Subscribe (SBPS) [5-8] are a class of publish/subscribe middleware defined by Guo in [4] as “any publish/subscribe mechanism in which the semantics of the message are used in the routing of publication to subscriber.” To be defined as using semantics requires additional knowledge to be gained from the user’s subscription or publication, and thus increases the likelihood that any content delivered will be of interest.

The subscriptions by a subscriber define the interest of the user, where SBPS allows messages to be routed towards subscribers, based not only on their specific subscriptions, but also on the semantic relationship between their subscription and the contents of the publications. This section looks at various implementations of SBPS, their characteristics, usage, advantages and disadvantages. Any SBPS aims to increase expressivity in the subscriptions and publications of network clients, whilst maintaining the efficiencies of publish/subscribe. Hence the key common characteristic of SBPS is increased expressivity, and is used as the common metric for comparison throughout this section. Before the State of the Art in Semantic-based Publish/Subscribe is discussed a sub-class of SBPS (Knowledge-based Networks), and three implementations of Knowledge-based Networking are introduced.

3.2.1 Knowledge-based Networks

Knowledge-based Networks (KBN) are a class of Semantic-based Publish Subscribe (SBPS) which must allow both content- and semantic-based subscriptions and publications to intermix. KBN subscriptions are formed from a *(Name, Operator and Value)*, whereas publications are formed from a *(Name, Value and Type)* whether semantic or non-semantic. This is the key identifier of KBN equal to CBN functionality extended with support for semantics in *Attribute Values* (publications) and *Attribute Constraints* (subscriptions).

3.2.1.1 KBNImpl

KBNImpl is an implementation of a Knowledge-based Network developed by Keeney and extended by Roblek in [31] as presented in [6]. KBNImpl is based upon the Siena CBN [2] specifically version 1.69 which extends the functions of the Siena CBN with the semantic operators, types and values introduced in the Background Chapter 2, Section 2.3.3. KBNImpl does not constrain which broker a client connects to and clients are free to choose any broker in the network topology, as is the case with Siena.

3.2.1.2 KBNMap

KBNImpl operates around a single source ontology, used for creating publications and subscriptions. The work of Guo [4] extended KBNImpl, forming KBNMap, offering support for

multiple mappings between semantic models and interlinking SBPS systems through custom gateways. In KBNMap, when a broker receives a publication or subscription, and if it does not hold the semantic model required to reason about the message KBNMap uses a probabilistic-based selection strategy to “dynamically select the appropriate mapping strategy” between multiple different semantic models (ontologies). Guo’s work links multiple semantic models, across multiple brokers, where required. It is important to note is that KBNMap does not constrain or cluster which broker a publisher or subscriber connects to and operates like KBNImpl in terms of the choice of broker a client takes.

3.2.1.3 KBNCluster

KBNCluster, a proto-type system developed as part of this thesis, combines KBNImpl with a policy system controlling, and methods for, clustering publishers and subscribers around brokers of similar interests. KBNCluster extends the code-base of KBNImpl, like that of Guo. The Design Chapter 4 and Implementation Chapter 5 will discuss KBNCluster in detail.

3.2.2 A semantic Infosphere

Turning the focus to other classes of SBPS a Semantic Infosphere [33] operates around and clearly defines one of the core principles of SBPS, that being, semantic knowledge delivery. It utilises an architecture where publishers publish semantic XML messages as an “Infosphere Information Data Object (IDO).” This IDO is subscribed to by subscribers, using message filters i.e. filters that bound the requirements of the subscriber against the IDO.

Infosphere operates around the principle of semantic templates, into which non-semantic messages are cast. Messages are cast into templates using the domain “hasTemplate,” the range being the specific template in question. Using such a casting algorithm sees each non-semantic message being cast into a semantic template, thus becoming pseudo-semantic.

The scenario, used in [33], gives an example as a battle tank-force. The message templates, used in this scenario, and into which all messages are cast are: “*Move; Event; Situation Report* and *Enemy Order of Battle*.” In each of the template classes there are multiple instances of message types, each with a domain and range. Message carriers (IDOs) pass across the network with attached semantic annotations that describe the content of the data object. Such attachment enables for expression of “both the annotations and the subscriptions in vocabulary from the common ontology” than allowing ontological reasoning against the ontological strand of knowledge in the publication against the query, or filter, stored as a subscription. This reasoning is used to calculate whether the ontological filter structure matches the publication structure and delivers the content if and when a match occurs. The matching mechanism, utilised within semantic Infosphere, is at its core the same as that used in the KBNImpl, which takes the approach of combining snippets of ontological messages with non-semantic messages allowing

both semantic and non-semantic messaging and matching to occur simultaneously albeit with casting. Where criteria outlined in the ontological representation of the publication match that of the subscription, then there exists a notification and this is delivered to the subscriber.

The work presented in Infosphere greatly supports the argument that the augmentation of non-semantic data, into semantic messages, increases the expressiveness of the publication to the subscription-matching algorithm and that semantics improve expressivity.

3.2.3 Semantic Toronto Publish/Subscribe System (S-ToPSS)

S-ToPSS, presented in [30], addresses the semantic matching problem between publication and subscription, proposing a three stage solution to the problem, as discussed below.

Firstly synonym (“a word or phrase that means exactly or nearly the same as another word or phrase in the same language”) translates “all events and subscription attributes with different names but with the same meaning to a root attribute.”

The second stage involves the placement of a subscription / event into a concept hierarchy where, for each new event, additional event entries may be added. An important aspect of the concept hierarchy is the subscription to and publication from an ontological class structure, populated with multiple distinct instances belonging to classes, thereby enabling for the delivery of subscriptions based on generalised / less specific notifications.

Thirdly a mapping function “specifies relationships which otherwise cannot be specified using a concept hierarchy or a synonym relationship.” This mapping function “correlates one or more **Attribute-Constraints** to one or more semantically related **Attribute-Value** pairs.”

The mapping process presented has one major drawback. It must be conducted by a domain-expert, is not automatic and requires that a domain experts mappings or representation of those mappings is correct and agreed upon. i.e. no other combination of mappings could be chosen. The implementation of S-ToPSS is justified in the following semantic matching example, in which the task is to match an employer to a prospective employee. The employer is looking for a candidate from a “certain university”, “with a PhD degree”, “with at least 4 years work experience.” This semantic query is matched to an employee who has a PhD, from a particular school, with at least four years work experience only because S-ToPSS is able to reason that “school” and “university” have the same semantic meaning, particularly in North America, through their equivalence in a knowledge-base via the common ontology.

3.2.4 Graphed Toronto Publish/Subscribe System (G-ToPSS)

An extension to S-ToPSS, G-ToPSS [34], was developed by the same authors as a graph-based publish/subscribe architecture for dissemination of RDF data. G-ToPSS works with a common architecture of publishing and subscribing clients laid across a broker, or a network of brokers.

G-ToPSS uses Really Simple Syndication (RSS) [23] web feeds as the source for content-dissemination which are pruned into RDF triples, where each triple is made up of a (*subject*, *object*, *property*). Each publication in G-ToPSS is seen as a “directed labelled graph” formed around simple RDF triples. Subscriptions are represented by five tuple events: “(*subject*, *property*, *object*, *constraintSet(subject)*, *constraintSet(object)*)”

The use of constraints can be applied to both subjects and objects where constraints are represented "as a predicate of the form (*?x*, *op*, *v*) where *?x* is the variable, *op* is an operator and *v* is a value." There are two available operators: "Boolean, for literal value filtering, is-a for taxonomy filtering." The class topology used within the G-ToPSS system enables an is-a relationship to exist between classes and instances, where constraints applied against this class structure enables a subscriptions and publications (formed from the taxonomy) to be matched against one-another.

Such an example is clarified when the taxonomy is visualized: G-ToPSS presents a hierarchical topology in which “Publication” is the parent to both “Journal” and “Conference Proceedings.” Subscriptions therefore can be formed around a publication being a conference paper and a content-based constraint such as “Year Published \geq 2008”.

This demonstrates, in G-ToPSS as in the KBNImpl, a clear mix of content and knowledge-based subscriptions and provides an interesting comparison to the KBNImpl, supporting, much like S-ToPSS did, the argument that the introduction of semantics into the publish/subscribe paradigm greatly improves delivery chances to the subscribing consumer supported through worked examples.

3.2.5 Semantic Message Middleware for publish/subscribe networks (SMOM)

The main aim of SMOM [35] is to provide subscribers with a more flexible way to describe their subscriptions. This is achieved using an adaption of the DARPA Agent Mark-up Language and ontology Inference Layer (DAML+OIL) which are RDF [20] / OWL [36] precursors.

This architecture is structured around a client-server model in which the client may be a publisher or subscriber. The server provides both an interface to the Java Messaging Service (JMS) layer and the additional semantic layer. The semantic layer is accessed via a client API used to route through the semantic layer, to the JMS layer where the semantic layer calculates semantic

relationships between classes and instances. This matching mechanism involves a “semantic topic matching function which is responsible for instance checking and inference between each subscriber’s class description and each publishers instance description.” Such a checking mechanism results in notifications being delivered to subscribers from publications, as and when they appear in the network.

In conclusion SMOM is shown to provide a valid argument behind the adaption of Content-based Networks to provide semantic services which in turn aids in the delivery of semantic content to users, who have previously registered interests.

3.2.6 An ontology-Based publish/subscribe System (OPS)

The authors of OPS [29] propose an architecture in which "the domain concepts in all events are integrated together to form a concept node, and the system matches events with subscriptions both semantically and syntactically." Implemented using RDF and DAML+OIL OPS describes the data model using a (*subject, object, property*) relationship in the form of RDF triples, where multiple triples form together to represent events and where by using a comparison of the two event models allows for subscriptions, in the form of RDF graphs, and publications to be matched against one another. OPS uses RDF graphs and ontological representation of knowledge represent a method of turning non-semantic networks semantic. OPS provides such a method for publish/subscribe semantic matching using RDF as well as fully evaluating the performance of the matching algorithm using mathematical proof. OPS provides a good balanced argument for proven performance, offset with the increased expressiveness of semantics.

3.2.7 Designing semantic Publish Subscribe networks Using Super-Peers (SPS-SP)

The work of SPS-SP [37] aims to integrate publish/subscribe in an RDF based peer-to-peer publish/subscribe system. The approach of SPS-SP supports advertisements, subscriptions and publications, where the broker network consists of two types of nodes: super-peers and peers. “Peers are typically network nodes which wish to advertise and publish data, and/or subscribe to data owned by others. A super-peer is a node with more capabilities than a peer (e.g more cpu cycles, power and bandwidth)." The super-peer backbone is therefore responsible for the processing of publications, advertisements and delivery of notifications to subscribers, based on their available and increased processing power.

In SPS-SP [37] a peer subscribes by sending a subscription to its defined super-peer access point. Once a super-peer receives a subscription, the super-peer processes the subscription, storing a local copy of the subscription in its subscription tree and also forwards this subscription to its peers. Much like Siena [2] the super-peers do not forward messages “which are subsumed by

previously forwarded subscriptions," this is referred to as covering and results in only newly registered subscriptions being forwarded to the super-peers neighbours, covered subscription being merged with subscriptions already held by the broker.

This approach to load sharing reduces some of the costs involved in processing and routing semantic messages across the network. By outsourcing message processing to more highly powered super nodes, the less powerful client nodes can access services provided by the super-peers without resulting in a degradation of performance.

3.2.8 iBroker

iBroker [38] aims to reduce the load placed on a user, when searching for knowledge on the web, by matching stored (semantic) user models to incoming publications.

The authors of iBroker propose using OWL based ontologies and SPARQL[39] queries, in place of the RDF and XML content-based approach. Using iBroker allows users to define "user profiles" which are matched against incoming ontology-based publications. If a match occurs, the publication is forwarded to the user as a notification.

iBroker uses an OWL parser, to extract (from incoming ontological publications) the classes and individuals associated with those classes, the properties, domain and range of those properties, all of which is stored in an adapted hash-map. The subscriptions, in the form of SPARQL queries, represent user profiles, and make it possible to compare stored subscriptions against incoming publications. In conclusion iBroker provides a good overview as to how RDF/OWL and SPARQL queries can be used in a publish/subscribe scenario, if only centralised, and with the costs involved in SPARQL/RDF message matching.

3.2.9 Analysis

3.2.9.1 Advantages of SBPS

Semantic publish/subscribe offer a number of benefits to the user over Content-based Networks. The use of semantic subscriptions and publications offer an increased level of flexibility, expressivity and a great level of meaning, in the notifications delivered to a subscriber.

The use of semantics in publish/subscribe aims to assure and increase the chance that users only receive messages which they have expressed a direct interest. Content-based Networks provide the user with a method of explicitly subscribing to content in which they express an interest via non-semantic *Attribute-Constraints*. In comparison, semantic subscriptions allow expressions of interest using semantic concepts, to be formed.

One benefit of any publish/subscribe model is the de-coupling that occurs between producer and consumer, connected only by a common broker or message delivery mechanism. This is true

except in the naming of operators and attributes, where agreement is required, as previously discussed. It has been shown through the introduction of richer semantic models into the publication/subscription matching process further increases the separation between content producers and consumers, via a linked content model.

However semantics can be costly in terms of operational performance, which must therefore be balanced against expressivity. Examples discussed as part of this section have established that semantics increase expressivity in publish/subscribe and thus it is argued that this increases the ability of a producer to reach a consumer, in as dynamic a manner as possible, across a network topology.

3.2.9.2 Disadvantages of SBPS

One major drawback to semantic publish/subscribe is the requirement for agreement between producer and consumer on the body of semantic information over which reasoning is to occur, pre-deployment. Whether such semantic subscriptions occur over an RDF graph or ontological model, or using some bespoke metadata requires the format and agreement of structured content to be agreed upon and loaded by clients and brokers alike removing some of the de-coupling and flexibility introduced through the use of semantics in the first place.

There are additional costs associated with SBPS which increase when compared to using Content-based Networks. These are associated with searching for semantic matches against incoming publications and stored subscriptions. The specific cost of using the various semantic and non-semantic operators are outlined in the Evaluation Chapter 6, Section 6.3.1 and shows that semantics dramatically increase subscription processing times and publication delivery times. An additional problem associated with the usage of SBPS is that the semantics need to be available across the network on each node and for each node to hold the same semantic model as its peers, hence the work of Guo [4] in KBNMap.

A common drawback within both content- and knowledge- and semantic-based publish/subscribe is in the use of acronyms or full text in the naming of publications or subscriptions. For example the subscription *name=Dominic* is not matched by the publication *FirstName=Dominic*. The problem of matching subscription to publication names is one that is outside of the scope of this thesis, but one that does exist and needs to be present in system designers / implementers.

In conclusion SBPS have shown to increase the expressivity of the publications and subscriptions matched against one another over a common semantic model. As will be shown in the Evaluation Chapter 6, Section 6.3, the use of semantics increases both expressivity and also the cost involved in matching publication to subscription. This thesis does not aim to reduce the costs involved in semantic matching of publication to subscription. However it aims to increase the chance that a broker, receiving a publication, will hold a matching subscription.

3.3 Publish / Subscribe clustering Techniques

Querzoni [16] defines the process of clustering, applied to publish/subscribe systems, as being akin to “subscription regionalism” where subscriptions matched by the same publications are hosted on nodes localized in the same region of the overlay network. In this section a number of approaches to clustering publish/subscribe systems are introduced and discussed.

Most of the systems reviewed as part of this chapter operate over an overlay network, a logical network built on top of an underlying IP network. In this thesis clustering aims to reduce the number of overlay hops between producer and consumer, not the geographical distance or necessarily the number of IP hops across the network. The aim of this thesis is to reduce the number of message brokers process events required in delivering a message. This has the benefit of reducing the number of overlay hops and so time taken to deliver a publication, thus reducing the processing load placed on the broker network and increasing subscription aggregation improving both processing overhead in individual brokers and the network as a whole.

Querzoni, in his survey paper on interest clustering techniques [16], compares the difficulty of clustering Content- to Topic-based Networks. Using a defined taxonomy of topics allows clusters of interest to be easily created around the topic taxonomy, and publishers/subscribers be assigned to those clusters. Therefore topic-based subscriptions and publications are very tightly bound to interests. Content-based subscriptions are however constructed from a more diverse combination of filters, in comparison to topic-based subscriptions, and therefore it is harder to calculate a suggested cluster for a content- or knowledge-based publication or subscription. This difficulty in clustering content- as opposed to topic-based is indicated through the greater number of clustered Topic-based Networks [12] [10] [13] [14] [15] than Content-based Networks [40] identified by Querzoni in his review paper [16]. This is attributed to CBN lacking an external structure used for forming publications and subscriptions (such as an ontology) that can be analysed and used for the purposes of anticipating commonalities and forming clusters.

This difficulty, in clustering content-based subscriptions and publications is somewhat reduced when semantic subscriptions are introduced. Much like the approach used with Topic-based Networks it again becomes possible to use the ontological structure of the ontology to classify clusters and most importantly calculate the central, semantic point of the user’s subscription or publication, when formed around semantic content.

The remainder of this section examines various clustering techniques within mainly topic-based and Content-based Networks; there is currently no known work to the authors knowledge on clustering SBPS systems or semantic query systems on such a scale.

3.3.1 Topic-based Publish/Subscribe Clustering

3.3.1.1 Boosting topic-based publish-subscribe systems with dynamic clustering (Tamara)

Tamara [12] introduces a “novel distributed algorithm that utilizes correlations between user subscriptions to dynamically group topics together into virtual topics (called topic clusters), and thereby unifies their supporting structures and reduces costs.” Central to Tamara is a reduction in operating costs through the grouping of clients who share similar interests around common brokers. In Tamara topic-clusters are formed from the “groupings of topics with similar sets of subscriptions into virtual topics.” Much like individual topics, virtual topics are allocated a unique channel identifier. Using these identifiers publishers can reach of all the subscribers of the topic cluster through that channel or topic-cluster.

Subscribers attach to the topic-cluster as follows: “a user declares their interest in a set of topics. The system then determines a subscription policy for that user, namely a set of topics and topic-clusters that covers the user's interests.” This step in the clustering process identifies the user’s defined interests and places the user based upon these interests into the correct topic-cluster. When a user’s subscription changes, the user repeats the placement process, resulting in a replacement of the user within a new cluster, or they stay in the same cluster. A similar approach is applied to publishers and the result is publishers and subscribers are clustered around common brokers of interest, such a principle is applied in KBNCluster, using SBPS.

In conclusion Tamara [12] provides for the grouping of pairs of individual topics to form new clusters; the addition of a topic to an existing cluster; the merging of two existing clusters into a single cluster, the removal of a topic from a cluster as well as the destruction of a cluster. This results in a system which is dynamic in the process of clustering, and the result is a low overhead system enabling the benefits of the clustering process to be evaluated through performance analysis.

3.3.1.2 Scribe: a large-scale and decentralized application-level multicast infrastructure

Scribe [10] operates around the principle of multicast, where one message is distributed to multiple members of an individual groups and can therefore be thought of as clusters of content or interests. Implemented using the Pastry [41] DHT, Scribe allows for a “fully decentralised peer-to-peer model in which each participating node has equal responsibilities” [10]. Being fully decentralised removes any requirement for central co-ordination on the creation, joining and management of clusters. Scribe provides an adapted model of topic-based publish/subscribe,

allowing users to subscribe to a multicast group and receive delivery of all messages sent to that group.

Using the Pastry DHT, Scribe acts as a layer providing API calls to create a group, join a group, leave a group and finally publish to a group. Each group is assigned a “single rendezvous” node that acts as an access point for communication with additional nodes, belonging to the group, where groups can be thought of as analogous to clusters. The use of forwarding, in Scribe, allows for messages arriving for delivery to a group to be both routed towards all members of that groups as well as allowing any other members of the group to route messages towards the other members of that group. This allows groups of clients who share similar interests to reduce the costs involved in routing messages from publisher to subscriber, again a process only possible in Topic-based Networks.

3.3.1.3 Data Aware Multicast (daMulticast)

Data Aware Multicast [13] (daMulticast) provides topic-clustering, using Topic-based Networks and is similar, in operation, to Tamara [12]. daMulticast uses a “decentralized multicast algorithm that is data aware in the sense that it makes use of information about the hierarchical construction of pre-agreed topics to dynamically create groups of interests around subscribers and publishers, according to topic hierarchies.” The authors of daMulticast [13] identify two main approaches to topic clustering, one focusing on publishers, the other on subscribers. The publisher-based approach creates a group for the publishers of a topic and for each individual topic. This is a relatively simple process where the topic-hierarchy is parsed and re-formed into topic-groups, which allow future publications to be routed towards the correct topic-group.

For subscribers, daMulticast creates a group for the subscribers of each topic aligning these subscriber topic-groups against those previously created for publishers. Event dissemination is conducted based upon a topic-hierarchy structure where a node receiving a message forwards that message to the super-peer for that topic-group. This node then forwards the message to all other nodes in its original topic-group. This continues until the path the message takes reaches the node at the top of the topic tree, the master topic. Along this path subscribers who are interested in the publication are delivered the notification. In conclusion, daMulticast operates as multiple topic sub-overlays, connected by super peer topic clusters which interconnect brokers and route publication between topic clusters, based around the topic hierarchy.

3.3.1.4 Topic-based Event Routing for peer-to-peer Architectures (TERA)

TERA [14] again only provides interest clustering in a topic-based publish/subscribe network. Each topic in TERA forms a topic-overlay, where different topic-overlays interact with one another through a general super-peer overlay. Subscribers join the overlay, representing their topic-interest and may be part of more than one topic-overlay at a time. The general super-peer

overlay is then used as a routing mechanism for messages from their source to the access point which links the general overlay to the specific sub-topic-overlay.

TERA uses a Topic-based Network and a distributed unmanaged P2P architecture, much like Tamara [12] for event dissemination. There are two overlays, the “global overlay” and the “topic overlays”. The global overlay is linked to the topic overlay(s) through topic access points; these access points are nodes on the global overlay, which are linked with all members of the topic-overly and provide access to and an entry point to a multi-cast group. When a topic access point receives a message, which is intended for the topic overlay they serve, the access point diffuses this message across the sub topic overlay, its cluster.

Subscribers join topic overlays, representing their interests. Publishers pass publications to the global overlay which routes these message across the global overlay until an access point (and topic overlay) is found for a matching topic and cluster. When passed to the topic access point the message (using a flooding approach) is diffused to all subscribers in the overlay. In conclusion TERA [14] proposes and evaluates a novel approach to the creation of clusters within Topic-based Networks using a global topic overlay linking multiple sub global overlays.

3.3.2 Content-based Publish/Subscribe Clustering

3.3.2.1 Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks

Sub-2-Sub (S2S) [40] clusters subscribers in Content-based Networks in contrast to the previous approaches using Topic-based Networks. S2S uses a fully decentralised P2P system where subscribers form clusters by comparing their subscription with subscribers around them using an epidemic approach and where there are no defined boundaries between clusters.

The epidemic approach used to form clusters operates as follows: “Each peer i maintains a reference to another node j . If s_i and s_j intersect, and this intersection is not yet fully covered by the subscription of another node to which i has a reference, they link.” Such an approach allows nodes to independently compare their subscriptions to those of one of their neighbours and create a link if they hold comparable subscriptions but only if there exists no other closer match across their view of peers. The intersection between subscriptions determines whether subscriptions match exactly, do not match at all, or match to some extent or another. Subscribers in S2S are organised into multiple rings of similarity across a single overlay. Publications are routed across the overlay using a greedy algorithm to cross the rings until they reach a matching ring, at which point the ring will act as a transport to other subscribers within that ring, delivering to them the notification. Three types of links between nodes are utilised in S2S [40] to assure the overlay does not self-partition into unreachable areas; these are:

1. Random links are created between nodes within the network.
2. Overlapping-interest links create mappings between rings that share common interests.
3. Finally ring links, between ring nodes, are used to distribute publications to subscribers.

The evaluation of S2S is conducted in terms of overlay construction; the cost involved in construction of the rings, inter-ring link communication, random link creation and overlapping links. In conclusion S2S [40] is one of the few examples of content-based clustering documented as part of this State of the Art review, similar to the approach discussed in KBNCluster.

3.3.2.2 Efficient Publish Subscribe through a Self-Organizing broker Overlay and its Application to SIENA

The work of Baldoni et. al. [28] is based on the Siena [2] Content-based Network focusing on the clustering of non-semantic publications and subscriptions aiming to “cluster brokers sharing similar interests in a limited number of overlay hops.” This aims to reduce both the load on the collection of overlay brokers, as well as the number of hops to route messages from publisher to subscriber. In the work of Baldoni et. al. the calculation of similarity between brokers within the network is calculated as the number of events matched on *broker Bi* against the number that would have been matched on *broker Bj*, where existing hierarchical links are followed first. Brokers compare views within one another, looking for similarities that have occurred in previously delivered messages. Based on these similarities brokers make decisions to either move towards one another, in the hierarchical broker network, or stay in their current location, connecting and disconnecting as needed. Baldoni’s system operates in a non-semantic Content-based Network and embeds the logic for clustering on each of the nodes in the network. Like S2S, Baldoni’s work presents one the few approaches taken to clustering Content-based Networks, using a comparison between the messages brokers have delivered, bringing together brokers with similar sets of interests. This is different to the approach in this thesis which takes a client-first approach to clustering.

3.3.3 Conclusion

In sections 3.3.1 and 3.3.2 a number of approaches to clustering publish/subscribe architectures have been discussed. These approaches have commonly been applied to Topic-based Networks, where the external topic structure can be utilised in the grouping of clients around clusters of interest. Content-based Networks have been shown to be more complex in terms of their suitability for clustering. This thesis aims to contribute to the State of the Art by introducing and evaluating the clustering of Semantic-based Publish/Subscribe networks through the use of the semantic model, used by publishers, subscribers and brokers alike. The next section of this chapter grounds the publish/subscribe middleware previously introduced, in a common taxonomy.

3.4 Publish/Subscribe Classification II

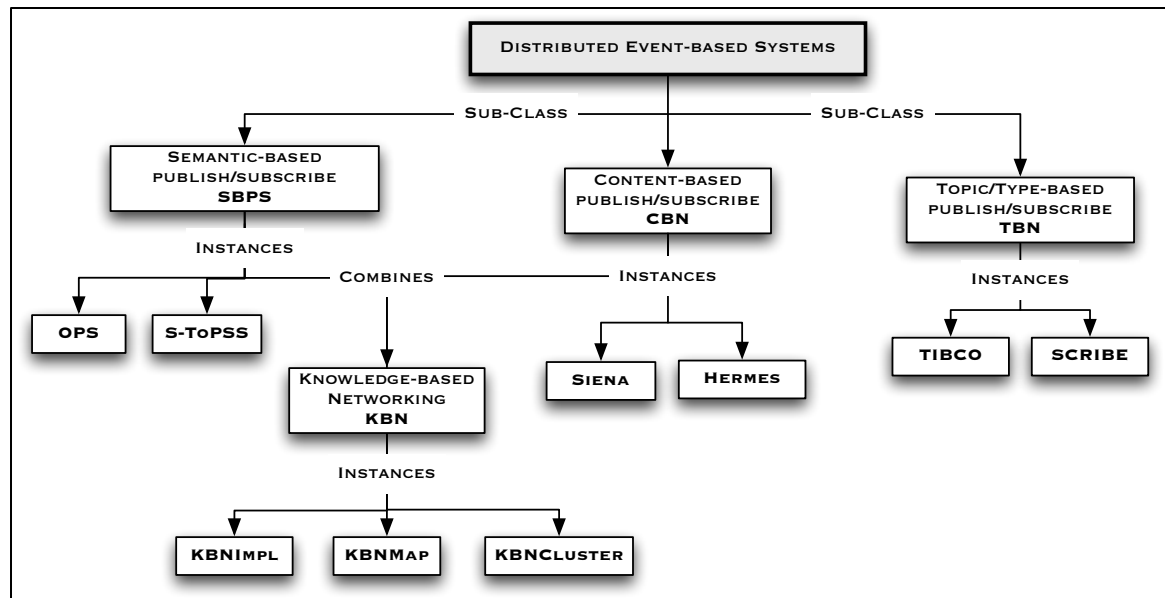


Figure 3: Classification of Publish/Subscribe Systems II

Figure 3 is the classification presented and previously introduced as Figure 2 in the Background Chapter 2, Section 2.3, extended to include a number of the implementations of publish/subscribe systems introduced in this Chapter. This figure only contains a number of example implementations, in each classification, not all those presented in this chapter. It is intended that this classification structure be used by Distributed Event-based System architects to classify their contributions in terms of the four major classes of such systems.

References to implementations above are as follows: **OPS** [29], **S-ToPSS** [30], **TIBCO** [1], **SCRIBE** [10], **SIENA** [2], **HERMES** [25] **KBNImpl** [6], **KBNMap** [4], Finally **KBNCluster** is introduced extended from **KBNImpl** in this thesis as the prototype implementation of clustering designed, implemented and evaluated.

Meier and Cahill [42] introduce a taxonomy of Distributed Event-based Systems which differs from the classification presented above in that theirs is used to describe any event-based programming system in terms of “a variety of properties including quality of service, mobility and security.” Meier’s taxonomy is for the technical classification of Distributed Event-based Systems, whereas the classification presented above, in Figure 3, is introduced to conceptually classify publish/subscribe systems within this thesis, and in future work, in terms of their placement into either topic-, content-, semantic-, and Knowledge-based Networks classifications.

The subsequent Design Chapter 4 of this thesis introduces **KBNCluster** as a managed approach to clustering publishers and subscribers around brokers, which match the interests of publisher, and subscriber, subsequently evaluated as supporting the research question and the objectives of this thesis.

3.5 Research Challenges gathered from the SoA

Three key common research challenges exist in the State of the Art, related to clustering publish/subscriber networks. In this section each of these challenges is discussed in turn.

***Research Challenge 1:** How can publishers who only attach transiently to a broker be clustered?*

In publish/subscribe systems (without advertising) subscribers define the scope of their interests before they consume content, in the form of subscriptions. Publishers send publications to a broker, but there is often no mechanism in place for a broker to subsequently contact publishers. Publications occur at the moment of their creation and then disconnect. This makes it difficult to define interests pre-publication. A solution to this is to use publication advertising [2] where publishers advertise descriptions of their future publications. They can then be instructed to connect to a different broker, and publish.

However in contrast to this the work, presented in daMulti-cast [13], proposes a solution for the clustering of both publishers and subscribers (but only in Topic-based Networks) where publishers only publish to specific topic-channels. By creating clusters around a finite set of channel definitions, brokers, publishers and subscribers are more easily clustered. The client or broker choosing their cluster based on a finite set of possible clusters formed around the same topic-channels, an approach that can be applied to Ontologies in KBNCluster.

***Research Challenge 2:** How can clustering exploit the semantic content of publications and subscriptions, in a semantic publish/subscribe system?*

It is inferred, from the range of systems evaluated in [16] and in the SoA in general that clustering based on topics is a task more easily achieved than clustering based on content-based messages. When clustering is based on topics, a publisher or subscriber joins a particular topic-cluster where topic-based subscriptions are seen as direct expressions of interest clients have expressed for future publications. For each topic a cluster can be created expressing and representing that topic.

Publishers, like subscribers, choose which topic and therefore which cluster they wish to reside in before they publish, using a pre-determined list. The issue with Content-based Networks is that of how to calculate and represent a common subset of the client using a **Name-Operator-Value-Type** subscription or **Name-Value-Type** publication. In content-based and KBN systems there is no finite set of interests (topic taxonomy) and clients do not express or place themselves in terms of their interests drawn across such a taxonomy. Interests (publications or subscriptions) are formed from the constructs of a (**Name-Operator-Value-Type** or **Name-Value-Type**) and matched against one another. However, the problem lies in calculating the similarity between two triples and forming clusters of interests from the complete set of clients across the network when triples are expressions of specific unique data values. This is a challenge addressed in SBPS by the introduction of an external semantic model of knowledge, used to assign clients to clusters.

Research Challenge 3: How efficiently can a single ontology be separated out into multiple sub-regions of separate interests?

The broker architecture, and the approach taken to overlaying clusters onto the broker network, used in this thesis, is based around a hierarchy that uses top-level brokers to receive messages with the broadest scope, allowing routers at the lower levels of the broker hierarchy to deal with more specific messages and those at the top the most general. In contrast to the approach taken in this thesis, where all brokers are seen as equal in terms of content they may route, [14], use a combined global and sub-topic overlay approach to clustering, where global overlays route messages from sub-topic cluster to sub-topic cluster seen as a ring (global overlay) with multiple topic-clusters extending from the global overlay. An initial, but later abandoned, approach taken in this thesis utilised an upper-level ontology, routing messages from cluster to cluster where each cluster dealt with a specific set of interests, a sub-ontology, like the approach taken in TERA. Therefore each cluster functioned around a partitioned, separate and smaller in terms of concepts and file size ontology with links between clusters that existed only at the top tier. It was planned that a top level global overlay of brokers would route messages from cluster to cluster, where one topic-level broker in the global overlay represents each cluster. Each cluster would use a separate, sub-section of the complete source ontology, where in contrast each global overlay broker would use the complete ontology.

However the difficulty in such an approach was in splitting a single ontology into multiple sub-ontologies, where each sub ontology represents a section of the main ontology and the original ontology. When a single ontology is separated into multiple sub-ontologies the new sub-ontologies become an inaccurate representation of only a part of the original ontology. An ontology is a model of knowledge, and although the sub-ontologies are valid ontologies in their own right, they are, once split, not an accurate representation of the complete original ontological model hence this approach was not carried forward.

3.5.1 Design Ideas gathered from the SoA

Drawing on the State of the Art several approaches are highlighted as having a direct bearing on this research to address the research objectives in Chapter 1. In this section those influences are discussed in turn.

3.5.1.1 Subscription Regionalism

One of the core ideas taken from State of the Art research in publish/subscribe clustering is the idea of “subscription regionalism” as defined by Querzoni et. al. [16]. Subscription regionalism, in topic-, content- and semantic- publish/subscribe is the placement of subscribers around brokers that share interests to one another. This results in fewer hops when delivering publication to

subscriber, less brokers involved in routing a message and a greater chance that a publication, arriving at any broker, will be matched to a subscriber of the same broker.

Subscription regionalism, as discussed in [16], is extended in this thesis and termed clustering, incorporating the benefits offered by subscription regionalism. A key difference between the two is that KBNCluster incorporates both the publisher and subscriber in the placement of clients across the network, whereas subscription regionalism, in [16], only deals with subscriber placement.

3.5.1.2 Ontology Partitioning

The work of Voulgaris et. al. in [40] present clusters not as defined boundaries of interest but as overlaying concept-spaces where clients are placed in the best region of the network for their interests. Placement within a concept space is calculated based on subscriptions to common concepts and the similarities in the naming of *Attribute-Constraints* and *Attribute-Values*. The principle of concept-spaces allow for parts of an ontology to be assigned brokers across the network, where publishers and subscribers can be placed into the clusters that best represent their interests or on the loose-boundaries between clusters.

In this thesis loose-boundary clustering is used to allow clusters of content to be applied over single or multiple brokers, whilst assuring that all publications will be delivered, to all interested subscribers, regardless of which broker receives them, as is the case in an un-clustered topology, but with the benefits of clustering. Using loose clustering, all brokers load and reason the whole ontology, regardless of their location in the network so they are able to process all subscriptions and publications as they arrive. No broker has any more knowledge than any of its peers. However loose-clustering still aims to focus the scope of publications and subscriptions received by each broker, in each loosely bound cluster.

Referencing the work of Guo and KBNMap in [4], tight-boundaries can be defined across broker routing based on specific, and separate ontologies. In KBNMap, when a broker receives a message and does not have the ontological knowledge to interpret the message, the broker loads mappings to other portions of a global overlay of brokers, where this overlay is constructed from multiple, semantically mapped ontologies interrelated by pre-defined explicit mappings. However, in contrast, in KBNCluster the most general clusters are assigned to the top-level brokers and at each lower level in the broker hierarchy more specific clusters of knowledge are assigned to brokers. However all brokers can route all messages / deal with any section of the ontological content. Such loose-boundary clustering allows for clients to be placed in the best region of the overlay by examining the set of semantically defined resources used in their publications or subscriptions.

3.5.1.3 Ontological Change

How often an ontology changes is driven by a change in the knowledge model, a change in perspective or simply new information arising and being incorporated into the ontology, as outlined by Flouris et. al. in [43]. However an assumption of this thesis is that the ontology will not change during experimentation. In KBNCluster therefore the clustering algorithm does not adapt the definitions, boundaries or placement of ontological clusters onto brokers over the period of experimentation, although the placement of publishers and subscribers does change, as their interests change. Clients' interests will change, but the underlying ontology will not.

In KBNCluster an additional feature is implemented which allows brokers and clients to be pushed an electronic ontological model and for this model to be reasoned over and loaded into memory. This feature was implemented so that brokers could be pushed new ontologies when the initial approach of multiple sub-ontologies, with a global upper ontology, was being investigated. This approach was made redundant when the difficulties in splitting a single ontology into multiple sub-ontologies whilst remaining the semantics of the original model. It is discussed here as an avenue of future work where research could be conducted into the process of updating ontological models on the brokers and clients of the network, as interests change, without requiring a full-restart.

However for the evaluated implementation presented in this thesis if the ontology does change the full system requires a cold-start. By adapting where users are placed within a static cluster structure assures that as users change their semantic interests this change is represented in their placement within a specific cluster. However the cluster topology is static, it does not change. What changes is the cluster a client is placed into, as the clients' interests change.

3.6 Conclusion

This chapter has provided an overview of current Topic- and Content-based Networks, which provide important reference architectures being the catalyst for the development of this research.

In addition to discussing Topic- and Content-based Networks, this chapter has examined the approaches taken by others in the application of semantics in publish/subscribe networks. The discussion as to how these technologies increase the expressiveness of subscriptions and publications is examined in each case.

This chapter has introduced topic- content- and semantic-based networks outlining the advantages / disadvantages of each as well as discussing architectural differences. In addition, a number of approaches have been analysed which apply clustering to mainly topic-, but also Content-based Networks. In general topic-based clustering of publish/subscribe networks creates a cluster per topic where publishers and subscribers are grouped around topic brokers that share common interests. In addition to the clustering algorithms applied to Topic-based Networks, there is also an examination as to the less wide-spread approaches applied in the clustering of Content-based Networks.

In conclusion, this chapter has related the research challenges of this thesis with the learning's drawn from the State of the Art. Design ideas are also gathered throughout the review and discussed. The next chapter of this thesis discusses the Design of KBNCluster, followed by a detailed documentation of the Implementation of the designed architecture.

4 DESIGN

4.1 Introduction

This chapter discusses the design of KBNCluster. This design is intended to explore the issue of clustering in Knowledge Based Network and of the dynamic management of such clustering. The design is an extension of an existing KBN implementation, KBNImpl. The design of KBNCluster can therefore be classed as an instance of Knowledge-based Network within the DEBS taxonomy presented in State of the Art Chapter 3, Section 3.4.

Dynamic semantic clustering as investigated in this thesis involves the ability to define the semantic centre (termed Medoid, see Section 4.3.1) of a client's interests making it possible to group clients more accurately around brokers sharing similar interests. A client's semantic centre (Medoid) is a single semantic entity that best represents the collection all their complete interests as expressed through the semantic subscription filters and publications they emit. In the case of KBNCluster, this entity is sourced from across an ontology representing all the concepts and their relationships active in the KBNImpl at that time.

In order to achieve clustering, a management approach is taken to the placement of publishers and subscribers across message brokers, where placement/clustering policies are rule driven and can therefore be manipulated by network operators as they change. Such patterns can only be understood through observing real world subscription and publication patterns and extracting their semantic attributes. This is not currently feasible with the present underdeveloped and state of SBPS deployment in real world applications. However, the impact of different management approaches and their processing and communication overheads on the generic operational behaviour can be readily assessed. It is for this purpose that the KBNCluster design is based on three core requirements for the operation and management of dynamic clustering, each discussed in turn, in this chapter. These are:

1. **Formation of clusters:** In KBNCluster an ontology is partitioned and overlaid across a number of brokers, assigning to each broker a portion of the ontological knowledge possessed by the whole network, thereby defining the semantic interests of that semantic content assigned to a broker. By matching a publisher's or subscriber's Medoid to the set of brokers the most appropriate broker, or cluster, for that publisher or subscriber, is estimated.
2. **Moving of clients and brokers across the overlay:** Being able to move clients to a more appropriate position in the overlay network is a requirement for clustering. KBNCluster instructs a client to reside in the most suitable cluster, automatically. However the interests of subscribers and publishers, in terms of content they subscribe to, or publish on,

and their suitable placement, may change over time. Hence the ability to move clients from cluster to cluster is vital for implementing dynamic clustering.

3. **Re-clustering of clients deemed to be in a sub-optimal cluster:** For each notification a client receives, the number of overlay hops the message has traversed during delivery can be calculated. From this it is possible to apply a re-clustering decision using the notification hop count metric to identify clients residing in a possibly sub-optimal cluster. If this occurs, a more suitable cluster for the subscribing client may exist and they may be suggested to move.

Having identified the core design requirements of KBNCluster, the high level design of the architecture is introduced, before a detailed example of the process of clustering is provided. Once both the architecture of KBNCluster and the clustering process have been detailed, the management system is discussed.

4.1.1 Extended and New Technology

The investigation performed with KBNCluster was made possible by the availability of an existing KBN design (based on the Siena CBN [2]), termed KBNImpl. KBNImpl is the work of Keeney et. al. as described in [6] and used in KBNMap [4] and this thesis as KBNCluster.

This section describes this technology, which has been extended to form KBNCluster as part of this thesis. These extended features are identified as technical contributions of this thesis, combining with KBNClusters evaluation, to address the research question and objectives outlined in the Introduction Chapter 1, Sections 1.1 and 1.2; KBNImpl has been extended with new core functionality, in order to achieve KBNCluster. The **extensions made to KBNImpl, as part of KBNCluster**, are outlined below:

- Subscriber and publisher Medoid calculation, vital for clustering clients around brokers of common interest termed clusters.
- A dynamic broker hierarchy, where brokers and clients can be moved around the topology, utilising a set of movement methods. These movement methods are implemented so that KBNCluster can migrate clients across clusters as their interests drift. Without these changes clustering would be static.
- Instrumentation of a hop count metric, embedded directly in notification message headers.
- Support for communication with brokers, publishers or subscribers through an adaptation of XML based KBNImpl configuration messages. This enables an unlimited number of policy server configurable messages to be sent from the policy server to KBNImpl brokers.
- The trigger broker provides for the policy server to subscribe to management messages and receive management updates, through matching publication to subscription.
- The ability for the policy server to communicate with publishers via the trigger broker and the publishers Universally Unique Identifier (UUID).
- Management Information Base (MIB) support for Managed Objects (MOs) where MOs are generic data models of brokers, publishers and subscribers.
- The combination of a rule engine and MIB query access in support of management policy execution.
- Provision for multiple actionable events, allowing executing policies to change where clients are placed across the network and request management information from them.
- A method for partitioning a single ontology into multiple strands of sub-ontologies subsequently overlaid onto a set of brokers, from the most specific cluster to least specific, top to bottom over the broker hierarchy.

In the next section the high level design of KBNCluster is presented and each component discussed in turn.

4.2 High Level Design

This section presents a high-level design, operational objectives and the design assumptions made in KBNCluster. Each of the constituent parts of the system is discussed in turn, providing support for the clustering of ontologies, placement of clients within clusters and re-placement of clients when identified as residing in the wrong cluster.

4.2.1 System Architecture

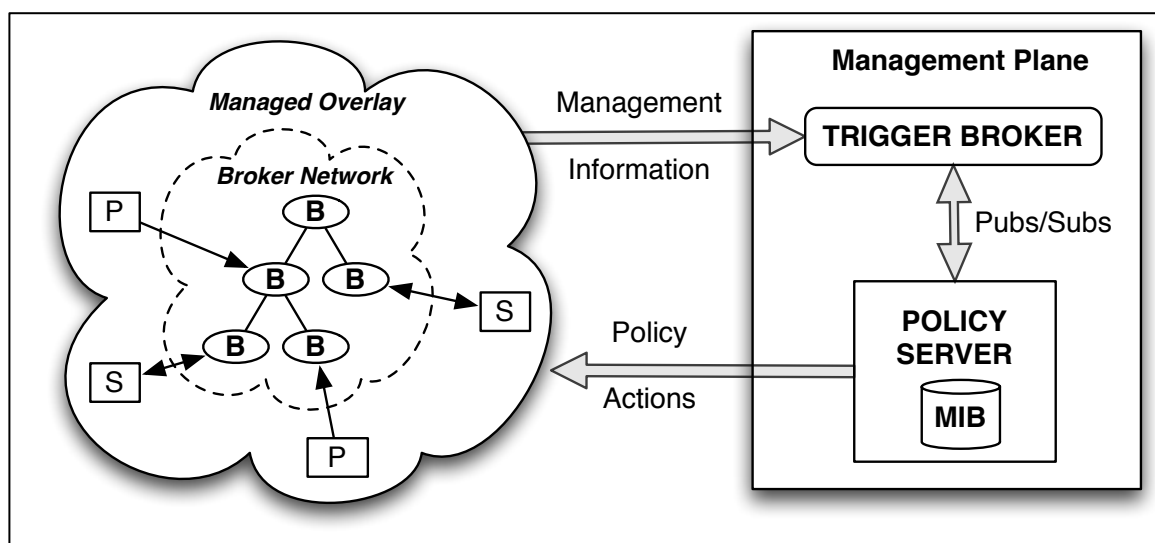


Figure 4: High Level Component Architecture

Figure 4 presents a high-level design overview of the main components, of KBNCluster. These components consist of the *broker network*, which combined with publishers (P) and subscribers (S) forms a *managed overlay* network. When this *managed overlay* is combined with the *trigger broker* and *policy server* they form the *management plane*, with potential to support a range of dynamic clustering in SBPS networks, which together forms KBNCluster. The trigger broker allows for subscriptions to specific occurrences of management data to be sourced from a large set of management source data, originating across the managed overlay. The filtering for relevancy of this data occurs in the trigger broker, not the policy server, removing this role from the policy server and placing this instead upon the trigger broker.

In this way the trigger broker acts as an aggregation and filtering point for management information, before it is delivered to the policy server as notifications. These notifications are used by the policy server to update or create Managed Objects (MO) entries in the Management Information Based (MIB), which are then used to trigger and evaluate policy rules. If a policy rule fires, i.e. the conditions in the policy are met, the action contained in the policy is enforced within the managed overlay, achieved either through direct messaging to brokers (and their attached subscribers) or via messages, sent across the trigger broker, back to publishers, addressed to their UUID.

4.2.2 Design Overview

The following section outlines the the approach taken in the design of the managed overlay, policy server and trigger broker in turn, central components of KBNCluster.

4.2.2.1 Managed Overlay Design Overview

Subscribers initially make an un-constrained choice about the broker they subscribe to but are then subsequently re-clustered by the policy server. This attachment process was kept as an operational characteristic of the KBNImpl, where a client initially attaches using an un-constrained decision and KBNCluster does not change this characteristic from KBNImpl. However once attached to a broker is a subscriber re-clustered.

Each broker, upon receiving a subscription from a new subscriber, passes this subscription and the subscriber's information to the trigger broker. The broker then waits for a re-clustering command to be received from the policy server and re-directs the subscriber to the most appropriate cluster, using a custom re-cluster instruction.

Brokers report to the policy server with appropriate management information as and when requested to do so by the policy server. These management messages include the number of subscribers, number of notifications delivered and broker's Medoid.

Publishers publish their Medoid to the trigger broker before they wish to publish. Once published they listen for clustering instructions from the trigger broker by subscribing to messages containing their Universally Unique Identifier (UUID). This UUID was required to identify each publisher in the system where there previously existed no way of identifying or communicating with publishers. By assigning each publisher a UUID, the policy server can communicate instructions to and receive management updates from the collection of publishers, notifying them of their new master broker and cluster or other operational instructions.

4.2.2.2 Trigger Broker Design Overview

The trigger broker receives subscriptions from the policy server and publications from the components of the managed overlay (brokers and publisher / subscriber clients) routing command messages from the policy server back to the publishers addressed, using their UUID. The trigger broker applies KBNImpl message matching against stored management subscriptions (originating from the policy server) and incoming management publications (originating from brokers and publishers in the managed overlay,) and delivers notifications, where a match occurs.

4.2.2.3 Policy Server Design Overview

The policy server initialises operation by subscribing to the trigger broker with subscriptions, representing its complete set of management interests. When notifications are delivered from the

trigger broker to the policy server these are stored, in individual MOs as part of the MIB. Once MOs is created or updated, the policy server executes the appropriate policies against the complete MIB. If policy conditions are met, then the policy server fires an action, which may involve notifying a broker or client in the managed overlay to adapt somehow, as outlined in the policy rule.

These objectives form the core operational characteristics of the components of KBNCluster, the managed overlay, policy server and trigger broker combining to form KBNCluster. The managed overlay performance, trigger broker and policy server are evaluated in detail in the Evaluation Chapter 6, Section 6.4.

4.2.3 Clustering Design Assumptions and Scope

Having discussed the high-level design objectives of KBNCluster in this section, some of the design assumptions and scope, applicable to KBNCluster, are outlined.

4.2.3.1 Publisher and Subscriber Design Assumptions

Subscribers choose a broker, an arbitrary choice, connect to that broker and then are re-clustered. The behaviour up until the point of re-clustering replicates the behaviour of KBNImpl. Once a subscriber has attached to a broker, the broker calculates the subscriber's Medoid and this is forwarded to the policy server, who calculates if this broker attachment represents the most suitable cluster for the subscriber, and if not which cluster would be more suitable. Once calculated, this cluster information is forwarded back to the broker that originated the clustering decision for the subscriber. The broker notifies the subscriber of its newly suggested cluster and the subscriber disconnects from its current broker and reconnects to the new cluster broker.

Publishers communicate, via the trigger broker, with the policy server, to establish where to publish based on their calculated Medoid. Once instructed, the publisher connects to the broker mandated by the policy server, and publishes their publications as in normal KBNImpl operation.

4.2.3.2 Design Assumptions around ontological clustering

Brokers load a set of ontologies during start-up. These ontologies are those used by publishers, subscribers and brokers to create publications or subscriptions. Ontologically unknown messages are messages that are received by a broker to which they do not hold the ontology required to reason. Such messages, as addressed by Guo [4] in KBNMap, are not present in any experiments conducted as part of this thesis nor is KBNCluster designed to deal with such messages. The research of Guo provides a solution to such problems.

Every broker, in KBNCluster, has access to the complete set of ontological knowledge required to reason about all semantic messages that it may receive. The approach taken here focuses on the

use of clusters that ensure publications and subscriptions are injected as closely as possible to one another in the broker overlay.

In summary, KBNCluster operates on the principle that a single source ontology is provided relating to the content that publishers, subscribers and brokers generate and route messages over. Through the use of Policy-Based Network Management (PBNM), operational characteristics of clustering change without requiring a re-start of the network. However if the source ontology changes then the full network, including all brokers, publishers, subscribers, the trigger broker and the policy server, need to be restarted. This restart is required so that the policy server can re-cluster the new ontology and the brokers, publishers and subscribers of the network re-load and reason over the new ontology; subsequently, each individual client is re-clustered appropriately.

4.3 Clustering Process

Having introduced the high-level design overview of KBNCluster, this section discusses the calculation of a client's Medoid and the clustering of brokers across the overlay network, where clusters are assigned to brokers as they are formed from a source ontology.

Once the clustering algorithm has passed over the source ontology, a number of clusters are created. These clusters are then assigned to brokers in the network and clients with interests matching these clusters attach as instructed by the policy server.

This section discusses how to place a client within a cluster and the identification of misplaced clients. Six sections are introduced, each key to the clustering algorithm evaluated in KBNCluster; these are:

1. The origins of the concept of the Medoid (Section 4.3.1).
2. Creating clusters around a source ontology (Section 4.3.2).
3. Mapping this source ontology onto an **A*** semantic map (Section 4.3.3).
4. Calculating a client's Medoid (Section 4.3.4).
5. Assigning a client to a cluster (Section 4.3.5).
6. Re-assigning misplaced clients (Section 4.3.6).

4.3.1 The Medoid

Before the clustering process is examined, this section introduces the concept of the Medoid as has been previously briefly discussed. The Medoid is used to represent the centre of the semantic interests of publisher or subscriber. It is calculated from the set of semantic (ontological) subscriptions or publications a client holds at a given point in time. It is not possible to calculate a Medoid, or similar single representation of interest, from a non-semantic subscription or publication, as they lack the structured graph of the ontology, from which the Medoid is calculated. The Medoid is represented by a single URL of an ontological class or instance taken from the source ontology on which semantic publications or subscriptions have been formed. It is deemed to represent the client's or broker's central point from a set of interests.

The source ontology is the ontology used by brokers, publishers and subscribers, in the KBNImpl, to express semantic messages, and is consistent with the name space properties of semantic web languages. In practice, the ontology present across the KBNImpl could be composed from several ontology documents taken from different sources, but containing at least one link connecting a term in each ontology document to one in another. However, this namespace partition does not impact on the Medoid calculation and is not therefore considered further in this thesis. Ontologies are contained in a single file throughout KBNCluster.

The term Medoid is first defined by Kaufman and Rousseeuw in [44] and discussed by Van der Lann, Pollard and Bryan in [45] where Partitioning Around Medoids (PAM) is presented as an algorithm used to calculate, given a source data set, the number of clusters and topological centre of those clusters. The work of Kaufman provides a basis for the concept of a KBNCluster Medoid in calculating clusters of similarities in large sets of human gene expression data. The major difference between the use of the Medoid in this thesis, and in Kaufman's work, lies in the PAM algorithm taking a complete set of input data and returning n multiple clusters and their respective Medoids, from across the complete data set, where n is provided to the algorithm.

However, the Medoid in this thesis is returned as the central point of the client from a set of interests.. The different approach arises from the operational need of KBNCluster in comparing the interests of a client to that currently held by different brokers (clusters,) and is seen as more suitable to a changing network configuration. This is in comparison to the PAM approach, which attempts to build a global model of semantic structure based on a single graph. In KBNCluster the Medoid is defined as representative of the point on a weighted graph with the smallest distance to all other points, within a query set. The query set is taken from all semantic elements contained in a publications *Attribute-Value (Name, Type, Value)* or subscriptions *Attribute-Constraints (Name, Type, Operator, Value)*, in effect the complete set of semantic elements referenced in either publication or subscription. Detailed examination of extracting the query set from publications and subscriptions is included in Sections 4.3.3.1 and 4.3.3.2 respectively.

In KBNCluster, the Medoid is calculated as the node (ontological URL) in a semantic graph of the source ontology (see Section 4.3.3) with the smallest combined distance to a given query set where the query set is a subset of the nodes in the graph. The semantic spread of a Medoid is also calculated. This represents the accuracy of the Medoid value. The semantic spread is calculated as the standard deviation from the Medoid, to each point in the query set. A smaller standard deviation for a given query set represents a more accurate Medoid, whereas a larger standard deviation indicates the Medoid has a less uniform distance to it.

4.3.2 Taxonomical Approach to Cluster Creation

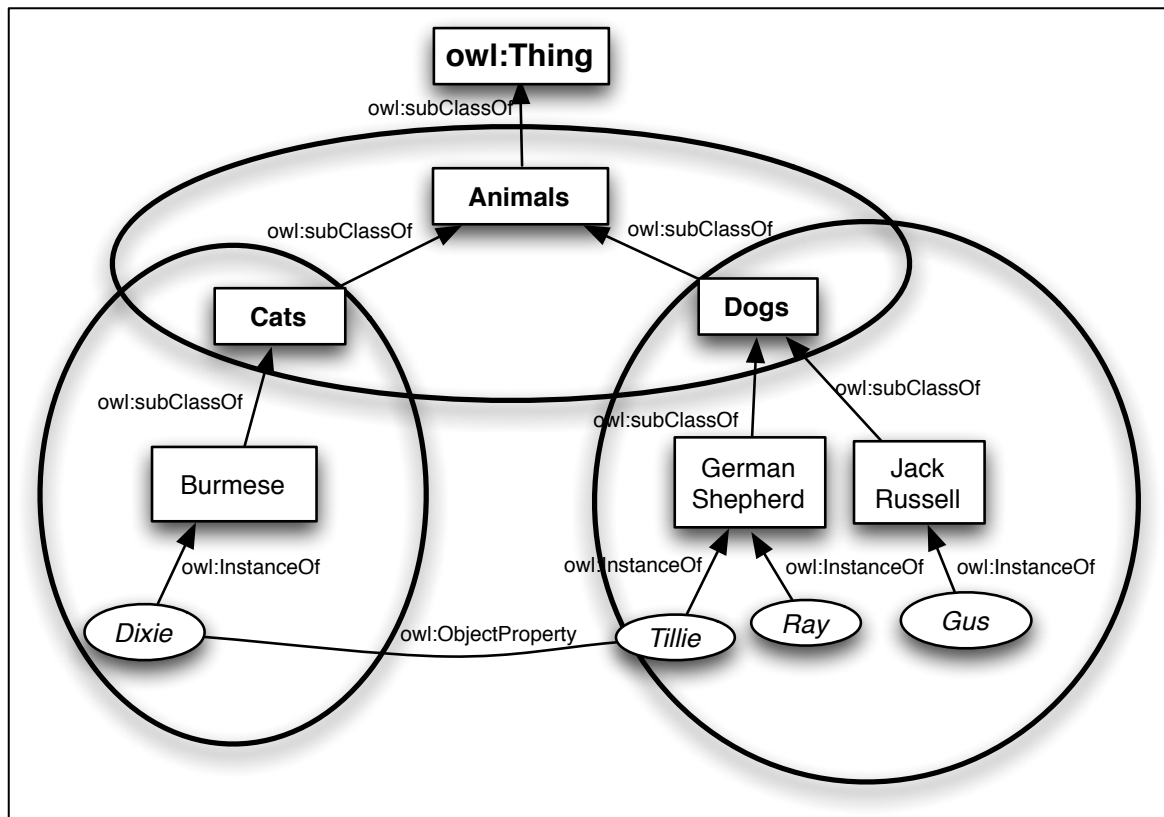


Figure 5: Sample Clustered Ontology

Figure 5 presents the sample “animal” ontology, previously introduced in the Background Chapter 2, of this thesis. The taxonomical cluster creation algorithm, used by KBNCluster, is applied to this ontology, and utilises only the structure of the ontology, the relationship between ontological nodes, in calculating clusters and not the labels of the elements, or their meaning. The classes at the bottom of the input ontology are presumed to be more specific than those at the top, as specific concepts are broken down into sub-concepts. Virtual-partitions of clusters, within a single ontology, as used in KBNCluster, do not physically split the ontology but only create virtual associations of interests, loose-boundaries, across the source ontology, which are then assigned to brokers.

Clusters are created using the following approach: for each top-level root ontological class, create a cluster of that class and its sub-classes. This creates the first *Animal* cluster, shown in Figure 5. Then algorithm then loops through each sub-class class of the top-level root classes and creates a cluster of that class and its sub-classes. This creates the two-second clusters *Cats* and *Dogs*, as shown in Figure 5. As each cluster is created the instances of each class, being clustered, are associated with the cluster. This process continues until the last sub/super relationship between classes, in the topology, is identified. For example there is no (*Burmese*) (*German Shepherd*) or (*Jack Russell*) cluster created; only the parent classes are formed into

clusters. This approach creates clusters at the highest level through to the lowest level of concepts, for a given source ontology.

After the clustering algorithm has been applied to the ontology, three distinct clusters have been created, shown in Figure 5, these are:

- The **Animal** cluster (inc. *Cats* and *Dogs*).
- The **Cats** cluster (inc *Burmese Dixie*).
- The **Dogs** Cluster (inc *German Shepherd, Jack Russell, Tillie, Ray* and *Gus*).

The cluster identifier for each is **Animal**, **Cats** and **Dogs**, where the concepts associated with each of those IDs are included in brackets. Once created for an ontology, these taxonomy clusters are overlaid onto the set of available brokers from the top of the broker hierarchy down to the bottom of the broker hierarchy. As the algorithm passes over brokers, the highest-level brokers in the topology are assigned the most general clusters (**Animal**), and the lower level brokers to the most specific clusters (**Cats** and **Dogs**). It is important to note that this algorithm is applied symmetrically it starts at the top of the ontology and moves from left to right across the ontological model. Once clusters are created the algorithm applies these clusters to the broker hierarchy from top to bottom left to right. Its loose association approach overlays the structure of the ontology across the topology of the broker from top to bottom, left to right.

This occurs due to the following principles of design: In a file held on the policy server, **brokers.txt**, all the available brokers are recorded from the top to the bottom of the broker hierarchy. On each line of the **brokers.txt** file is a broker lower down the hierarchy of all brokers in the KBNImpl deployment. Using the principle that the brokers at the top of the hierarchy are at the top of **brokers.txt** and those at the bottom of the hierarchy at the bottom of the file enables cluster assignment from most to least specific from the top to the bottom of the broker hierarchy. A by-product of this process is that if there are more brokers than there are clusters, these clusters will be assigned to only the top-level brokers in the hierarchy, typically those with more resources and less constraints. Once this process is complete the policy server assigns the remaining number of clusters from the top-down across the broker network, until all clusters are assigned to a broker, and each broker may hold one or more clusters.

A key feature of the architecture is that the clustering algorithm is a modular function and thus replaceable. The algorithm described here attempts to provide an efficient solution for using clustering to improve network performance, specifically in terms of the hops taken to route publications, as will be evaluated. Yet the clustering algorithm could be replaced with an updated or modified approach, providing for future adaptations, as experience in the forms of ontologies and the client interests that are expressed in them, become better understood.

4.3.3 Creating an Ontological A* Map

Having clustered the source ontology, this section of design discusses the calculation of a client's Medoid. This is achieved using an A* graph of the ontology [46]. A* is a variation of the Dijkstra graph search algorithm [47] which provides the shortest path between any two nodes in a weighted graph of nodes. These nodes are connected via weighted edges, and this graph is used in Medoid calculation from a set of input query nodes, the complete semantic interests of publisher or subscriber.

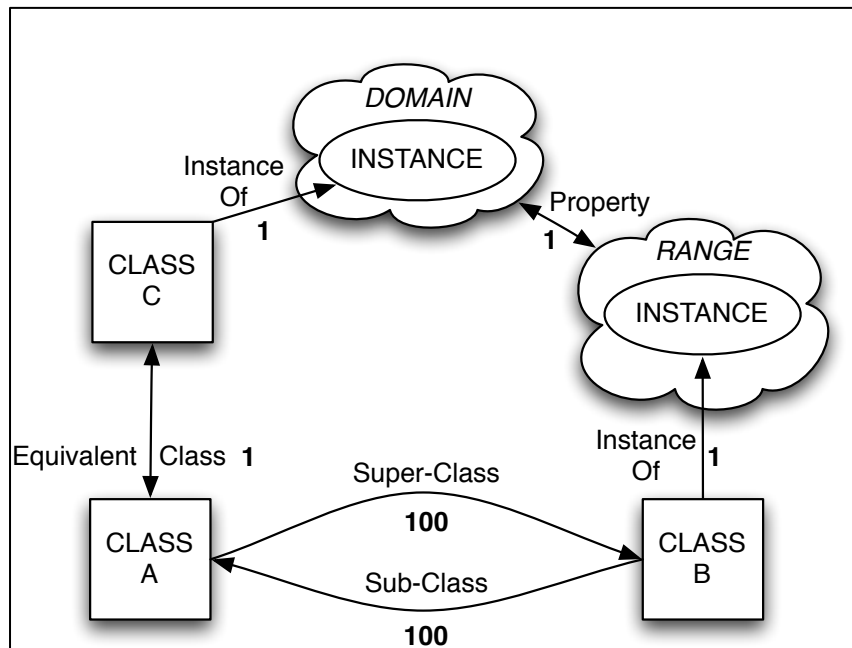


Figure 6: A* Ontological Map

Shown in Figure 6 are the example relationships between ontological elements (nodes) and their edges (weighting) in a generic model of ontology, used in transposing a source ontology onto a semantic A* map. In KBNCluster the cost to travel between nodes are as follows:

- Sub Class to Super Class = 100 / Super Class to Super Class = 100.
- Class to Equivalent Class = 1 / Equivalent Class to Class = 1.
- Class to Instance (One Way) = 1.
- Domain Instance to Range Instance = 1 / Range Instance to Domain Instance = 1.

The algorithm allows for alternate weightings to be given to the arcs between ontological concepts, between 1 and 100; the example weightings shown above used in the evaluation of KBNCluster. These weights were chosen to represent the broad relationship between classes in the super – sub class relationships and the closer relationship between instances and classes. It is suggested future user studies are conducted where ontology experts are asked to weight edges between nodes and averages of the user-suggested values used in future experimentation. These values are configurable in KBNCluster, here they are suggested values, user-sourced variables should be established to fully validate these values.

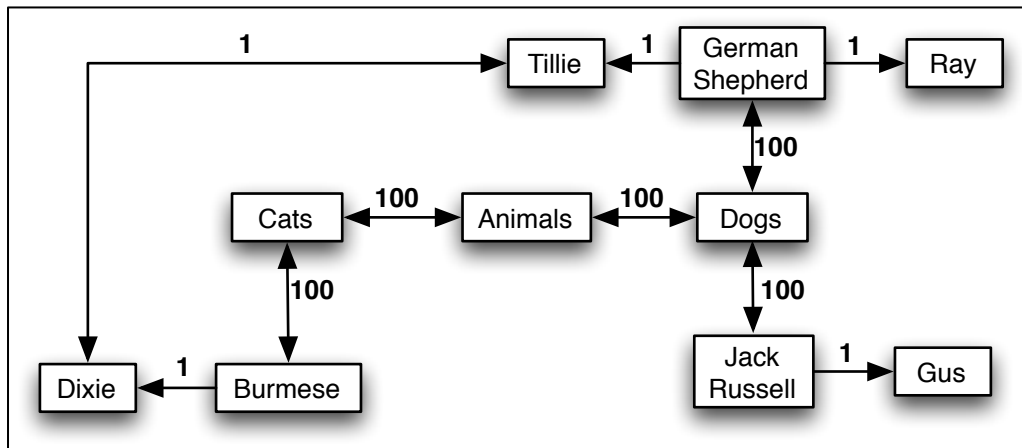


Figure 7: Sample ontology graphed using A* algorithm

Such weightings are applied in Figure 7 to the example animal ontology plotted onto an **A*** map of nodes and edges, using the previously introduced rules and weights. The next two sections look at extracting the semantic query set from publications and subscriptions required for calculating a clients Medoid.

4.3.3.1 Extracting Semantic Elements from Subscriptions

When calculating a subscriber's Medoid, multiple *Attribute-Constraints*, combined as subscription filters, are used. Each attribute constraint is made up of a (*Name, Type, Operator, Value*). Using these *Attribute-Constraints* the following approach is applied in the calculation of a subscriber's Medoid, using their set of semantic subscription filters.

1. For each subscription from a given subscriber, the broker extracts into an array all the semantic *Attribute-Constraints* from that subscriber's subscriptions, therefore including all the ontological concepts referenced in the subscription.
2. Calculate and return the Medoid for this collection of concepts, see Section 4.3.4.
3. The broker passes the Medoid to the trigger broker, which forwards it in a cluster decision request to the policy server.
4. Awaits re-clustering instructions for the subscriber, from the policy server.
5. The broker informs the subscriber which then, if necessary, moves itself to the assigned cluster.

In summary, the broker calculates Medoids for newly attaching subscribers, and these are sent to the policy server in the form of a single Medoid and its associated semantic spread via the trigger broker as a subscriber information message. Instructions for clustering subscribers, are then sent back to the broker, and these are forwarded to the relevant subscriber.

4.3.3.2 Extracting Semantic Elements from Publications

Calculating the publisher's Medoid is a more challenging task than that used in subscriber Medoid calculation. Subscribers issue subscriptions to brokers and this allows the broker to act as

an intermediary between the subscriber and policy server in clustering the subscriber. However, publishers are not persistent. They appear, publish and disappear. Due to this the following approach is applied in clustering publishers:

1. The publisher client calculates their own Medoid, based on their publications **Attribute-Value** formed from a **(Name, Type, Value)**.
2. Publishers forward their Medoid to the policy server via the trigger broker including their UUID. It is important to note the trigger broker is a single point of failure in KBNCluster.
3. The policy server calculates the most suitable cluster for the publisher, and sends these instructions back to the publisher, addressed to their UUID, via the trigger broker.
4. The publisher connects to the assigned cluster/broker and publishes.

In summary, due to the transient relationship publishers have with a broker, publishers calculate their own Medoid before publishing and are clustered. In contrast subscribers have their Medoid calculated by the broker to whom they initially subscribe. Subsequently the subscriber, with all their subscriptions, is moved to the suggested cluster.

4.3.4 Medoid Calculation

This section presents a description of the process of calculating a Medoid using the principles outlined in the previous two sub-sections and utilising a selection of query nodes, provided as ontological URIs, calculated from a subscription (**Attribute-Constraints**) or publication (**Attribute-Values**).

In the first step of Medoid calculation, the shortest distance, ds , is defined between all nodes n and n' in the **A*** map by the **A*** algorithm. ds is calculated between all nodes in the ontology, formed into an **A*** map, based on the configured weighting for each different arc in the ontology.

Next, the shortest distance dq is calculated between each node n in the semantic map, and the complete query set q , using ds to calculate the collective cost from each node to the query set. Once calculated, dq is equal to the node with the shortest sum of all distances to the complete query set. The ontological URL associated with that node is returned as the client's Medoid.

If the query set only contains one ontological entry, then the Medoid for that query set is automatically returned, being the single entry of the query set.

4.3.4.1 Example Medoid Calculation

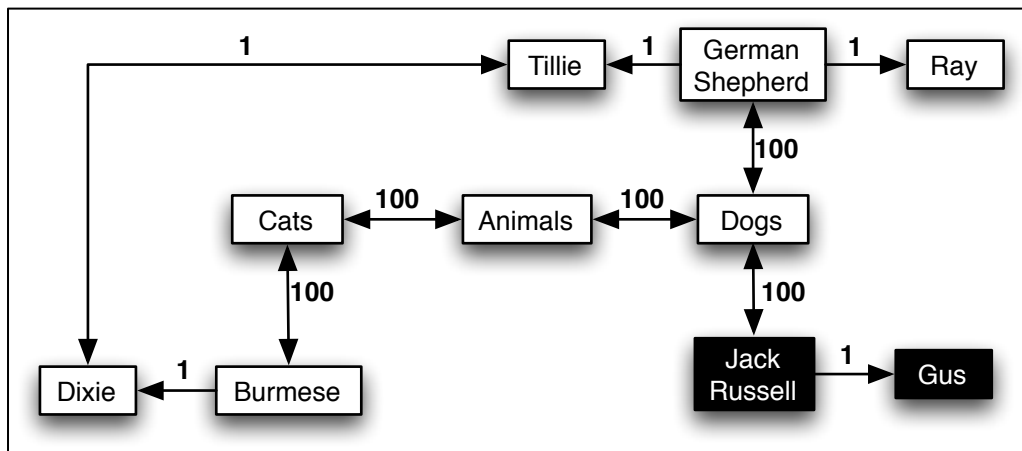


Figure 8: Example Medoid Calculation

Shown in Figure 8 is a Medoid, calculated from the example “animal” ontology previously introduced. In the above example, the query set has been identified as containing references to both the ontological class *Jack Russell* and instance *Gus*, bolded only for clarity.

The graph is searched for the node with the shortest distance to each entry of the query set. The Medoid calculated from the query set is *Jack Russell* with a cost of one, zero to itself and one to *Gus*, remembering the Medoid can itself be part of the query set, or outside the query set. Therefore, from the query set, *Jack Russell* is calculated as the Medoid for the query set, and thus the suggested cluster *Dog* returned to the subscriber or publisher as the calculated cluster for the query set and placement. In the next Section 4.3.5 each possible Medoid, shown above, is placed into a cluster, using KBNClusters algorithm for cluster placement and the previously defined clusters.

4.3.4.2 Summary: Medoid Calculation

When an ontology is plotted onto a weighted graph of nodes and edges, using the rules described in this chapter, distances between ontological concepts can be calculated. Combined with this, a publication or subscription is made up of elements from an ontological model. Given the concepts referenced in a publication or subscription (query set) and a weighted graph of the source ontology, it is possible to calculate the ontological point which represents the shortest distance (possibly any one of the ontological concepts) to the combined query set.

The cost is calculated from every point in the weighted graph to the entire query set. The point in the weighted graph with the smallest combined cost to travel to each of the points of the query set is deemed to represent the Medoid of the client, even if this point resides outside or inside of the initial query set. In KBNCluster weighted graphs are used to calculate the shortest path between two points.

4.3.5 Cluster Placement

At this stage a collection of clusters has been calculated, semantic publication or subscription elements extracted, and a single semantic interest (Medoid) of publisher or subscriber calculated. This section discusses client placement into a suitable cluster.

An initial approach was taken using the WordNet:Similarity tool [48] to measure the semantic similarities between the Medoid concept and each of the clusters. Each cluster is represented by a set of strings (class or instance, etc) extracted from the ontology. The WordNet similarity API was used to calculate the similarity between the Medoid string and each of the strings in each of the sets of clusters, the cluster with the highest combined similarity to the Medoid being returned as the suggested cluster for the client. However, this approach assumes that the usage of word in the ontology definition corresponds to the definition in WordNet or could be found in WordNet in the first place. These are not necessarily valid assumptions, especially if the ontologies were more product oriented, or written in completely different languages. Therefore KBNCluster took a taxonomical, not a synonym-based, approach in calculating the cluster most suited to a query Medoid, as follows:

1. Given a client Medoid, as a URL, all clusters are searched to see if they contain that URL.
2. The cluster that the Medoid URL resides in is the cluster where the client should be placed, and this is returned to the client as their suggested cluster. A URL (ontological element) in KBNCluster can only be contained in one cluster.

Summarising the above: each client's Medoid is a URL, calculated, from the source ontology, and represent a client's semantic centre. Every cluster definition is similarly a collection of URLs from the source ontology, partitioned into clusters of interests. Searching the collection of clusters, comparing each URL in each cluster to the Medoid URI of a client, a match, when found, returns a cluster (and so the associated broker associated with that cluster to the subscriber or publisher.)

For properties the class applicable to the property's domain is chosen for the suggested cluster. If the property has no domain or range, then it will not have previously been mapped onto the **A*** semantic map. If it had been mapped (lacking a domain and range) it would have been disconnected from the rest of the map and therefore the cost of travel to the property would be infinity and thus be excluded. In KBNCluster each broker is assigned a single cluster in the first pass of the algorithm. If more clusters exist than brokers those brokers at the top of the broker hierarchy are assigned the remaining clusters. One cluster can only be assigned to one broker but one broker may have more than one cluster assigned to it.

The cluster calculation algorithm is run, using each Medoid from the source ontology shown in Figure 5, and estimation as to their correct cluster given. If no cluster is returned, for any reason, then the subscriber stays connected to the broker they initially attached to, and publishers make an

arbitrary choice of broker and publish. Examples of Medoid matches and cluster placement from the previous example ontology are as follows:

- **Class** Medoid estimations:
 - Query Medoid *Animals* should be placed in the cluster *Animals*.
 - Query Medoid *Cats* should be placed in the cluster *Cats*.
 - Query Medoid *Burmese* should be placed in the cluster *Cats*.
 - Query Medoid *Dogs* should be placed in the cluster *Dogs*.
 - Query Medoid *German Shepherd* should be placed in the cluster *Dogs*.
 - Query Medoid *Jack Russell* should be placed in the cluster *Dogs*.
- **Instance** Medoid estimations:
 - Query instance *Gus* should be placed in the cluster *Dogs*.
 - Query instance *Tillie* should be placed in the cluster *Dogs*.
 - Query instance *Ray* should be placed in the cluster *Dogs*.
 - Query instance *Dixie* should be placed in the cluster *Cats*.
- **Property** *Tillie* chases *Dixie* and should be placed in the cluster *Dogs*.

Five ontologies have been run against the cluster placement algorithm and these are presented in the Appendix B.

4.3.6 Re-clustering

Over a period of time the interests of a subscriber may drift, and, as this occurs, so will their Medoid will change. The policy server provides a number of policies, allowing the network manager to set the interval for a subscriber or broker to report their Medoid back to the policy server. Once requested, and received, this updated Medoid is used to calculate whether the client is still in the right cluster or whether the client needs to be re-clustered to a more suitable cluster.

In reaction to this drift, the policy server uses a numerical and objective measure of assessing when clients require re-clustering. To adopt a simple but operationally significant parameter for such a policy decision, the number of hops over which a notification passes in being delivered from publisher to subscriber was identified. Each subscriber is responsible for calculating its mean hop count from all delivered notifications, and this is used in determining if re-clustering is required. This hop count information is included in a notification report, requested by the policy server from the subscriber, delivered via the Trigger Broker.

Once the policy server receives a notification report for a subscriber, if the mean hop count is deemed too high, they are identified as requiring re-clustering. Metrics for determining too high a hop count are outlined in a configurable policy. If a subscriber is identified as requiring re-

clustering, its Medoid is passed to the same process that is used in their initial clustering with their updated Medoid. If a new more suitable cluster is found, the client is requested to move.

Key to re-clustering subscribers is the requirement that their semantic interests (subscriptions) change, which results in a change to their Medoid and thus re-clustering. With regard to publishers, they only ever request to be re-clustered and are never directed to re-cluster. For this reason it is required that publishers request to re-cluster at a defined interval, or after their publications change and they identify this change. Publishers are only ever clustered in their first publication into the network and then identify themselves as requesting re-clustering. Additionally, and because of this, publishers calculate their own Medoid, and do not have it calculated by the broker they are attached to, i.e. they by-pass brokers and deal directly with the policy server. There is no metric which can be used for identifying publishers in an un-suitable cluster.

Finally, it is possible that an application node could be both a publisher and a subscriber. In such a situation the application is clustered, in KBNCluster, based upon their subscription (subscriber) Medoid and not their publication (publisher) Medoid.

A proposed solution to such a problem is to initiate 2 client interfaces, 1 for publishing and one for subscribing, where each client interface, held by a single application, would be resident in different clusters. This is seen as an application level decision and thus outside the scope of the objectives addressed in KBNCluster, and not evaluated.

4.4 Trigger Broker

Having established the clustering process, this section discusses the design of the trigger broker. The trigger broker acts as an intermediary between the managed overlay and the policy server, responsible for filtering incoming publications from the managed overlay and delivering relevant messages to the policy server. Shown in Figure 9 is a high level overview of the internal operations of the trigger broker.

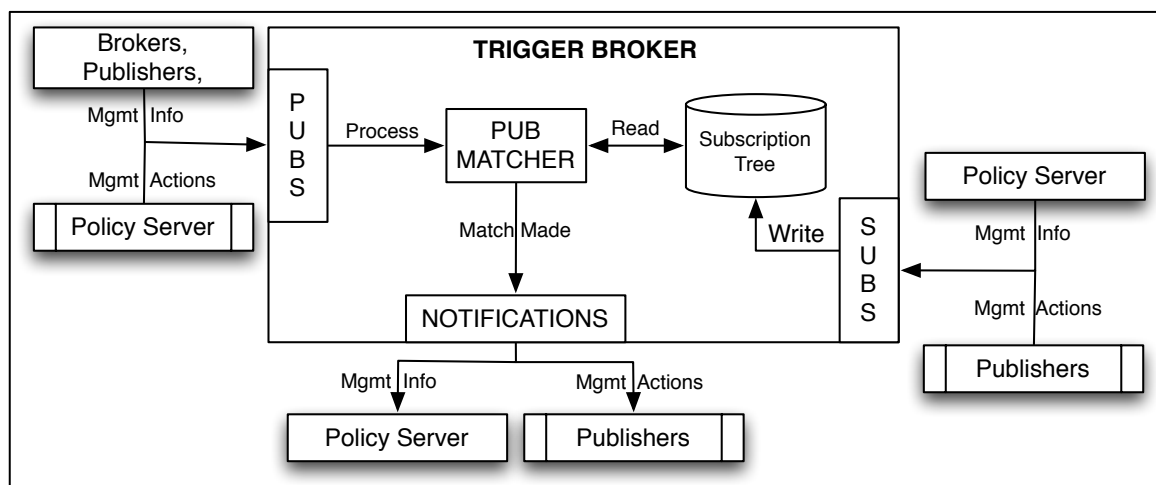


Figure 9: Trigger Broker, High Level Overview

The subscriptions and publications sent to the policy server from the managed overlay via the trigger broker are based on a management ontology designed specifically for representing management information sourced from the network (i.e. the managed overlay). It therefore is distinct from the ontology used in the core of KBNCluster and is not part of the source ontology used by the brokers and non-management clients in forming clusters. This management ontology is shown in Figure 10 and is used to form semantic subscriptions on a range of management information published by components of the managed overlay.

There are two ontologies used in KBNCluster. The first is the *content* ontology this being the ontology used by KBNImpl nodes for producing publications and subscriptions. This content ontology can be thought of as the messaging ontology, on which semantic publications and subscriptions are formed, this ontology is present and used in KBNImpl.

The second ontology is the *management* ontology this is the ontology used to tag management messages. All nodes in KBNCluster use both the *content* and *management* ontology, the content ontology used for delivering publications and subscriptions and the management ontology used in delivering management messages between nodes in the overlay, the trigger broker and policy server. This ontology is not present or used in KBNImpl.

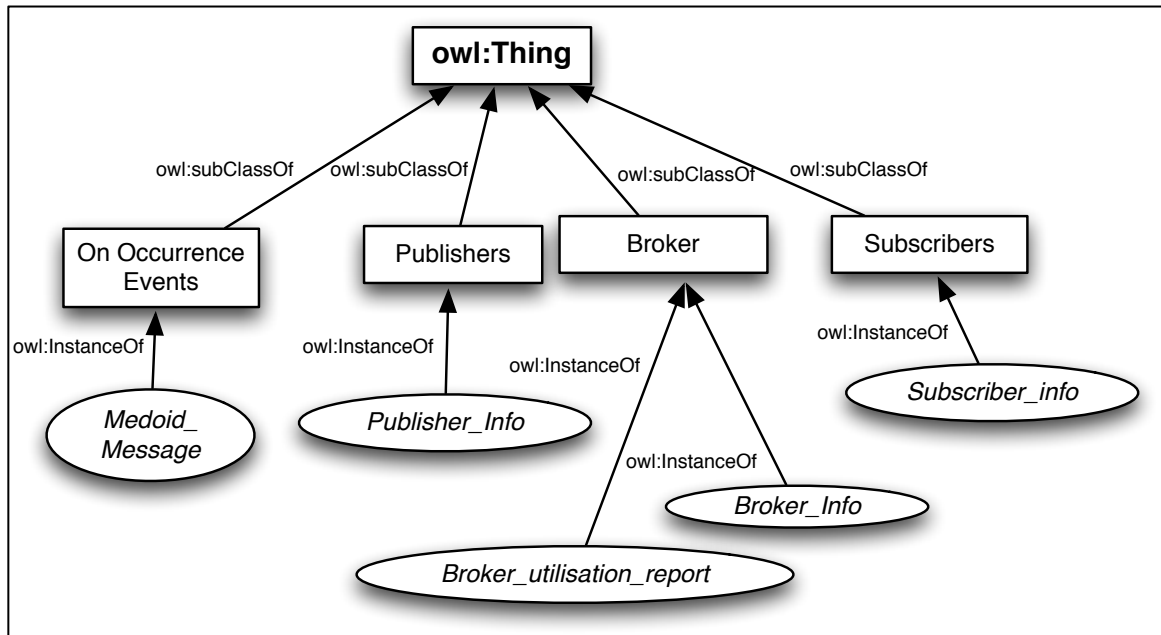


Figure 10: Trigger Broker Ontology

By using the trigger broker ontology, it is possible for semantic subscriptions to be formed around management information needed by the policy server, accurately filtered by the trigger broker. Exploiting existing KBNImpl functionality in this way represents a new approach to the semantic delivery of management data, using semantic message matching. In the next section, the types of subscriptions, sent into the trigger broker in KBNCluster, are discussed.

4.4.1 Subscriptions to Trigger Broker

There are a number of types of subscriptions that can be issued by the policy server to the trigger broker for management information based on the trigger broker ontology, these are:

1. Medoid messages (From brokers, publishers and subscribers).
2. Broker information messages.
3. Subscriber information messages.
4. Publisher information messages.
5. Broker utilisation reports.

An example semantic technique for subscriptions to management data allows a manager to subscribe, using the ISA operator to a class of instances. Again using the ontology in Figure 10, some examples are shown in Code Example 1, and publications shown in Code Example 2.

Type	Name	Operator	Value
<i>OntInstance</i>	<i>MessageType</i>	<i>ISA</i>	<i>Broker</i>

Code Example 1: Example Managerial Subscription II

Type	Name	Value
<i>OntInstance</i>	<i>MessageType</i>	<i>Broker_Info</i>
OR		
<i>OntInstance</i>	<i>MessageType</i>	<i>Broker_utilisation_report</i>

Code Example 2: Example Managerial Publication II

The example above allows for subscriptions to specific management events, or classes of management events to be issued to the trigger broker by the network manager. It is also possible, but not evaluated in this thesis, to couple these semantic filters with low-level data filters that, when applied to the content of the messages, provides both semantic and non-semantic subscription mechanism, e.g. where data properties have a value over a certain threshold.

4.4.2 Publications Received by the Trigger Broker

This section describes publications from publishers and brokers that are subscribed to by the trigger broker, these include (used in KBNCluster):

Medoid Messages, (Code Example 3), requested by the policy server from brokers and contain the broker's Medoid(s), Medoid standard deviation(s), master broker (if any), broker address and port.

Type	Name	Value
<i>String</i>	<i>BrokerID</i>	<i>KBNImpl Broker Address</i>
<i>Instance</i>	<i>BrokerMedoidMsg</i>	<i>Ontological Instance URL</i>
<i>Instance or Class</i>	<i>Medoid</i>	<i>Ontological URL</i>
<i>Integer</i>	<i>NumOfSubs</i>	<i>Number</i>
<i>Double</i>	<i>StdDev</i>	<i>Number</i>
<i>String</i>	<i>Master</i>	<i>KBNImpl Broker Address</i>
<i>Integer</i>	<i>Port</i>	<i>Number</i>
<i>String</i>	<i>Receiver</i>	<i>Local KBNImpl Broker Address</i>

Code Example 3: Example Medoid Message

Broker Info Messages, (Code Example 4), received by the trigger broker on broker start-up, or as a broker is moved across the network. They contain the broker's master and receiver address in the form of *tcp:IP_Address:Port*, used by KBNImpl for message delivery.

Type	Name	Value
<i>Instance</i>	<i>BrokerInfoMessage</i>	<i>Ontological Instance URL</i>
<i>String</i>	<i>Master</i>	<i>KBNImpl Broker Address</i>
<i>String</i>	<i>Reciever</i>	<i>Local KBNImpl Broker Address</i>

Code Example 4: Example Broker Info Message

Subscriber Info Messages, (Code Example 5), received by the trigger broker each time a subscriber attaches to a broker or moves to a new broker and contain the subscriber's master broker, Medoid, unique KBNImpl subscriber ID and their complete subscription as a string entity as generated by the KBNImpl code-base.

Type	Name	Value
<i>Instance</i>	<i>SubscriberInfoMessage</i>	<i>Ontological Instance URL</i>
<i>String</i>	<i>Master</i>	<i>KBNImpl Broker Address</i>
<i>Instance or Class</i>	<i>Medoid</i>	<i>Ontological URL</i>
<i>String</i>	<i>Subscriber ID</i>	<i>KBNImpl Subscriber ID</i>
<i>String</i>	<i>Subscription</i>	<i>Full KBNImpl Subscription</i>

Code Example 5: Example Subscriber Info Message (Publication)

Publisher Info Messages, (Code Example 6), received by the trigger broker from publishers before they publish. They contain the address, their current master broker in the network, previous Medoid and UUID.

Type	Name	Value
<i>Instance</i>	<i>PubInfoMsg</i>	<i>Ontological Instance URL</i>
<i>String</i>	<i>CurrentMaster</i>	<i>KBNImpl Broker Address</i>
<i>Instance or Class</i>	<i>Medoid</i>	<i>Ontological URL</i>
<i>Integer</i>	<i>StdDev</i>	<i>Number</i>
<i>String</i>	<i>PublisherID</i>	<i>KBNImpl Publisher UUID</i>

Code Example 6: Example Publisher Info Message (Publication)

Subscriber Notification Reports, (Code Example 7), requested by the policy server and contains the subscriber's unique KBNImpl subscriber ID, master broker and a notification map. This notification map contains the subscriber's mean hop count, standard deviation associated with the mean and a map of the number of occurrences of the number of hops for all delivered messages. For example if a subscriber has received (6 notifications across 1 hop), (2 across 4 hops) and (1 across 7 hops) their resulting map would be formed as follows: (*NotificationMap*="(1-hop=6, 4-hop=2, 7-hop=1)".)

Type	Name	Value
<i>Instance</i>	<i>NotificationReportMessage</i>	<i>Ontological Instance URL</i>
<i>String</i>	<i>Subscriber ID</i>	<i>KBNImpl Subscriber UUID</i>
<i>Integer</i>	<i>HopCountMean</i>	<i>Number</i>
<i>Integer</i>	<i>StdDev</i>	<i>Number</i>
<i>String</i>	<i>NotficaitonDeliveryMap</i>	<i>Array of Hops and Counts</i>
<i>String</i>	<i>CurrentMaster</i>	<i>KBNImpl Broker Address</i>

Code Example 7: Example Subscriber Notification Report (Publication)

The five publications shown above are extendable. However these are in the form used in the evaluation of KBNCluster in Chapter 6. A key benefit of using such message formats is that the policy server is able to subscribe to certain semantic occurrences or specific low-level data values occurring in the set of management data. In this way the trigger broker acts as a central rendezvous point and broker for management messages, where these messages can be filtered using subscription filters incorporating semantic and non-semantic constraints that directly represent the conditions present in management policies in the policy server. Therefore the

messages received by the trigger broker and the data stored in the MIB can be tuned to the minimum needed to support the active policies or operational policy.

4.4.3 Interactions between KBNImpl Publishers and the Trigger Broker

In addition to receiving status reports from the brokers of the network (and their subscribers), the trigger broker acts as an intermediary between the policy server and publishers in receiving and sending instructions back to publishers via their UUID. This again uses the publication and subscription feature of the KBNImpl, with the publisher and the policy server acting in both roles to exchange messages in both directions. The approach required for communicating with publishers, is outlined below:

1. Publishers connect to the trigger broker and subscribe to messages containing their Java generated Universally Unique Identifier (UUID).
2. The publisher publishes a Publisher Information Message (PIM) containing both its UUID and their Medoid.
3. The PIM is delivered to all interested subscribers, via the trigger broker.
4. The policy server uses the PIM to calculate the most suitable cluster for the publisher. This instruction is published back to the trigger broker including the UUID of the publisher to whom the instruction is addressed.
5. When the trigger broker receives clustering instructions from the policy server addressed to a UUID (of a publisher), The Trigger Broker notifies the relevant publisher (the only node to have subscribed to management messages with its UUID).
6. The publisher receiving this clustering decision from the policy server, via the trigger broker, attaches to the broker identified in the clustering decision, publishing content as normal.

Full sequence diagrams of this communication flow can be found in the Implementation Chapter 5, Section 5.4. However before this the next section of this chapter discusses the policy server's operation, addressing MIB/MO design, load balancing, policy authoring and actionable events, before this chapter is concluded with an overview of the key characteristics of the design of KBNCluster.

4.5 Policy Server

The policy server allows for managerial commands to be enforced across the managed overlay. This is achieved using six core principles of operation, these are:

1. The policy server subscribes to the trigger broker to receive notifications of management information.
2. The policy server listens for and receives notifications from the trigger broker.
3. When the policy server receives notifications from the trigger broker, the MIB updater finds, or creates, the MO for the node identified in the message.
4. Once a MO has been created or updated, the policy engine runs the complete set of policy rules over the complete MIB, to assure concurrent changes to the MIB are caught. The complete set of policies, 4 in total, defined in section (4.5.2) are executed each run.
5. If a policy fires, the policy engine calls the actioner, enforcing the policy outcome in the Managed Overlay.
6. Once all actions have been completed, the policy server returns to point two above, awaiting notifications of new, management information.

Shown, in Figure 11, is the internal architecture of the policy server, where numbered components are linked with the numerically bulleted descriptions presented above.

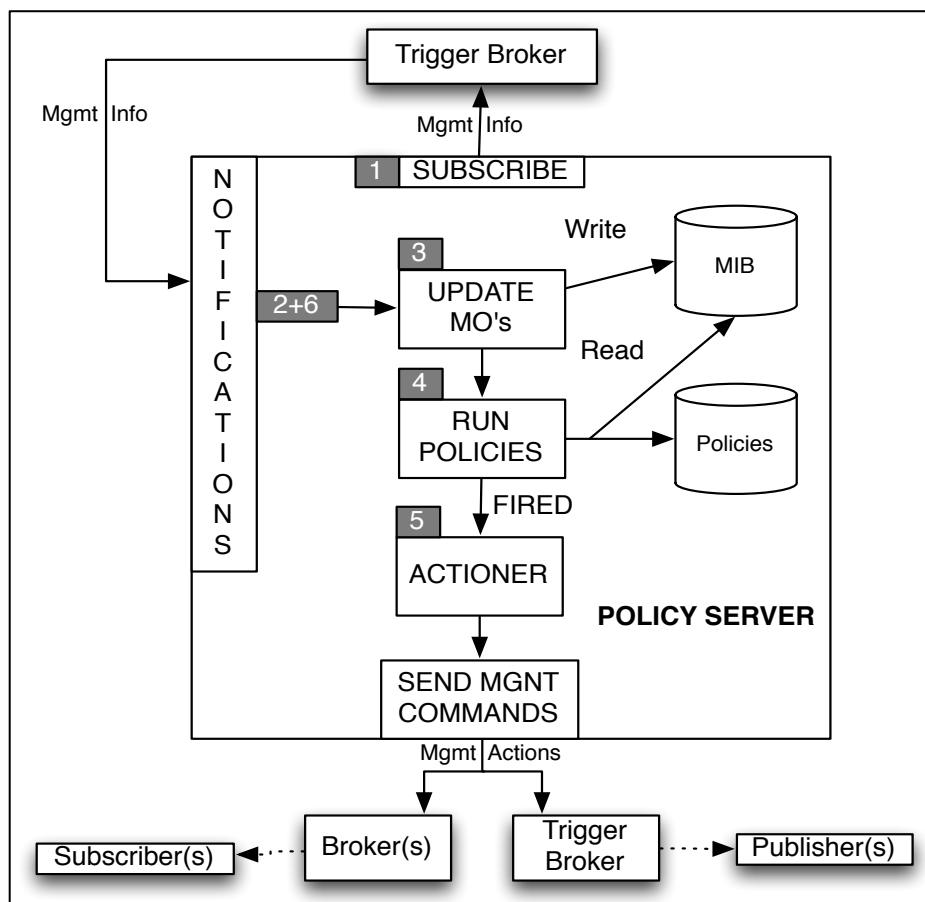


Figure 11: Policy Server, High Level Architecture

The policy server has been designed for experimental purposes in the evaluation of KBNCluster and is not production ready or designed to scale to real-world deployment levels. Section 6.4.1.2 in the Evaluation chapter evaluates the costs and upper limits of the policy server as implemented in KBNCluster. These boundaries are found to be acceptable for the purpose of evaluating KBNCluster but not scale to real-world deployments. Therefore re-engineering of the policy server is required before real-world deployment according to the design ideas presented in this Design chapter.

4.5.1 MIB/MO Design

The Management Information Based (MIB) is constructed from a number of Managed Objects (MO) where each MO is a simple Java data storage object. This enables the variables from incoming notifications (via the trigger broker) to be parsed and stored in the correct MO for the node that published the management information. Overlay broker MOs are identified by the broker ID, publisher MOs by their UUID, and subscribers by their KBNImpl subscriber ID, each of which are unique in any given KBNImpl deployment. Details of the MIB and MO design are included in Section 4.5.1 with full UML class diagrams being presented in the Implementation Chapter 5, Section 5.6.3, however high level descriptions are presented here.

Broker MO's contain:	<ul style="list-style-type: none"> • Unique KBNImpl broker ID, • Broker's master and their own network addresses, • Number of currently active subscribers, • Number of currently active subscriptions, • Medoid of the broker and the Medoid's standard deviation.
Subscriber MO's contain:	<ul style="list-style-type: none"> • Unique KBNImpl subscriber ID, • The broker address to which the subscriber is currently attached, • Medoid of the subscriber and the Medoid's standard deviation, • Notification map, stored as a hash map where the key is the hop count and the value is the number of occurrences of each hop.
Publisher MO's contain:	<ul style="list-style-type: none"> • Unique KBNImpl publisher UUID, • The broker to which the publisher is attached, if any, • Medoid of the publisher and the Medoid's standard deviation.

Table 9: Broker, Subscriber & Publisher MO Key Elements

Shown in Table 9 are the three types of MO used in the MIB, by the policy server. These MOs are generic Java objects with a range of variables and associated 'getter' and 'setter' methods. Each MO acting as a buffer between incoming management messages and the policy engine, by storing relevant management information to support subsequent policy condition execution.

The remainder of this section looks at the high-level policy objectives, applicable to these MOs as they arrive at the policy engine, and finally the set of policies that are run over the MIB objects in the evaluated KBNCluster.

4.5.2 Policies

As has been previously discussed, policies are constructed using an **Event-Condition-Action** form. The **Event** part of the policy rule is based on a publication arriving at the policy server, being processed, and stored into an existing or newly created MO. The **Condition** part of the policy is based on filtering conditions that can be applied to data stored in the MO. The **Action** is to be carried out if the **Condition** is satisfied, given a certain **Event**. This section looks specifically at the high level goals of the policies held by the policy server.

Policy 1: (Subscriber Placement) occurs when a subscriber appears for the first time or is identified as requiring re-clustering and outlines the conditions, across the overlay that should be met before a subscriber can be clustered. *These policy conditions are the number of brokers in the network and the number of clusters.*

Policy 2: (Publisher Placement) occurs when a publisher appears for the first time or is identified as requiring re-clustering and outlines the conditions, across the overlay that should be met before a publisher can be clustered. *The policy conditions are the number of brokers in the network and the number of clusters.*

Policy 3: (Notifications Reports) determine how often the policy server should request notification reports from each subscriber and sets out the metrics that subsequently identify subscribers who are deemed to be in the wrong cluster. *The policy conditions are the number of brokers in the network, number of subscribers, number of clusters, mean hop count and mean standard deviation of the hop count.*

Policy 4: (Broker Medoid Reporting) determines how often a broker should report their Medoid and their Medoid's standard deviation. This information is used to identify the broker to be clustered, and then the specific subscribers to be re-clustered. *The policy conditions are the number of brokers in the network, number of subscribers and the number of clusters.*

Policies 1 and 2 deal with clustering publishers and subscribers across the network, whereas policy 3 deals with the identification of a subscriber placed in the incorrect cluster. Policy 4 is provisioned for, but not utilised, in the evaluated KBNCluster. Rather, based on management performance evaluations described in the Evaluation Chapter 6 Section 6.4, a client-first approach to clustering is taken, where KBNCluster monitors and moves subscribers first, publishers second and leaves the broker network static, not moving or adapting the Broker hierarchy.

There are two reasons for the exclusion of policy 4: firstly, the Siena CBN that was extended to form KBNImpl and used in KBNCluster is intrinsically static in the connection of brokers to one another; secondly, the cost of collecting and clustering brokers, using their management data, as will be shown in experimentation and evaluated in the Evaluation Chapter 6, Section 6.4.2, is costly and high. The use of policy allows the choice of how clustering design is set by a network manager and changed over time. However this choice is provisional, based on management

performance described in the Evaluation Chapter, and could be easily changed in future implementations, e.g. if this aspect of performance required improvement.

Such flexibility is a key design goal in adopting a policy based management approach and is supported by the KBNImpl-based mechanism for communicating between the management system and the managed overlay represented by the KBNCluster implementation.

4.5.3 Actionable events

Although many of the actions discussed as part of this section have been mentioned previously, they are repeated here for clarity and as a summary of KBNCluster's overall management operation.

The policy server provides a set of actions to the policy engine, that when called, allow certain policy outcomes to be enforced across the broker network. These form the *Action* part of the *Event-Condition-Action* loop. Given an *Event* and a matched set of *Conditions* the policy engine chooses an *Action* from the actioner to be enforced in the identified broker, publisher or subscriber. This section discusses the actions available to the policy server.

1. Subscriber clustering (input: sub master, sub ID, sub Medoid)

Given a subscriber's ID and Medoid, this method calculates the cluster that most suits a subscriber (as discussed in Section 4.3.4). Once the suggested cluster of the subscriber is calculated, the broker, to whom the subscriber in question is currently attached, is directly notified of the change. This broker then instructs the subscriber to move. This action is sent directly from the policy server to the unique KBNImpl address of the broker to whom the subscriber is attached, and not via the trigger broker.

2. Publisher clustering (input: pub ID, pub Medoid)

Given a publisher ID and Medoid, this method calculates the cluster that most suits a publisher (as discussed in Section 4.3.4). Once calculated, the action message is published to the trigger broker, addressed to the UUID of the publisher to be clustered; the publisher, having previously subscribed to future publications containing its UUID, receive this as a notification and connects to the broker identified in the message and publishes as normal.

3. Broker clustering (input: broker ID, broker Medoid)

If instigated, not the case in the evaluated version of KBNCluster, the policy engine calculates the broker to be clustered. This broker is identified as having the largest Medoid standard deviation. Once called, this method calculates which of the broker's clients should be clustered. KBNCluster provides for the clustering of brokers but however does not enforce this, as is discussed in the Evaluation Chapter 6, Section 6.4.2.

4. Notification Reports (input: trigger broker address, interval)

This method instructs brokers to command their subscribers to report to the trigger broker with notification reports. These reports are sent as publications directly to the trigger broker at a scheduled interval outlined in policy.

5. Broker Medoids (input: trigger broker address)

This method instructs brokers to report their Medoid report to the trigger broker. These reports contain their Medoid, the Medoid's Standard Deviation and the broker's clients associated with each of their Medoids. Similar to action 4, these messages are defined to be sent at an interval specified in policy.

6. Publisher Medoids (input: trigger broker address)

This method instructs publishers to report back their Medoid reports to the trigger broker at an interval outlined in policy.

In concluding this section of actionable events and their association with the policy format:

1. The **event** part of the policy comes from MO updates / creations on the MIB.
2. The **condition** part of the policy is outlined in the policy rules, and compares the MO being queried against the policy condition.
3. The **action** part of the policy is provided through the policy server's actioner.

4.6 Key Design Characteristics and Conclusion

KBNCluster allows for and provides for intelligence outside of the network, so that managerial goals are not hardwired into the network, but designed, contained and executed by the policy server, enforcing actions across the managed overlay, as required. This is critical in Semantic-based Publish/Subscribe networks, where performance is dependent on the patterns of semantic interest expressed in subscriptions and publication, which in turn may shift across of the network. These patterns and their shift cannot be reasonably predicted in advance and must therefore be understood through the monitoring of the operational behaviour of clients using the network. Therefore, the flexible, policy-based management solution adopted in KBNCluster is essential to the viability of future of clustering Semantic-based Publish/Subscribe deployments.

In supporting such external management, notifications of management data are pushed to the policy server, as they occur, by the nodes themselves. This is done via the trigger broker, using the publish/subscribe functionality of KBNImpl as opposed to being pulled or requested from each node using a dedicated management protocol. The use of subscriptions for the gathering of management information also supports the potential for a multiple manager architecture, where multiple managers manage multiple overlays, across a network of interconnected trigger brokers.

In this chapter, the clustering process applied to the managed overlay using the trigger broker and policy server, has been introduced. Each has been discussed in terms of design requirements, design decisions made and operational rationale. In the next chapter the prototype implementation of KBNCluster is discussed. In summary, the key characteristics of the design of KBNCluster are documented as follows:

1. **Formation of clusters:** The process of clustering an ontology into multiple clusters of interest has been outlined in a sample ontological model. This has been achieved by utilising the semantics built into the structure of the ontology, as opposed to the labels of the ontological components. In this approach, clusters have been formed using loose boundaries between semantic concepts, ensuring any broker, anywhere in the network, can process messages on any sub-set strand of the ontology, but still has an ontological specialisation in terms of the content they are optimised to route.
2. **Subscriber Clustering:** Subscribers are clustered via the broker they initially attach to, negotiated between the broker and the policy server, the subscriber is instructed to move to the most suitable cluster.
3. **Publisher Clustering:** Publishers communicate with the policy server, via the trigger broker, in a pre-publication, phase. Once a cluster is suggested, the publisher connects and publishes, checking for a more suitable cluster, as required. A self-identifying approach.

4. **Re-clustering:** The principle of re-clustering has been introduced in terms of collecting notification hop count statistics and updated Medoids from subscribers, deciding which to re-cluster based on these values.
5. **Managed Overlay:** The managed overlay has been introduced as consisting of the message brokers, publishers and subscribers combined with and connected to the trigger broker and policy server.
6. **Trigger Broker:** The trigger broker has been introduced as a medium, connecting both the managed overlay and the policy server receiving subscriptions from the policy server and routing management information to interested subscribers, one case of this being the policy server.
7. **Policy Server:** The policy server has been introduced and documented as comprising of the message processor, MIB/MO updater, policy engine and actioner. Each operates in union with the policy server to receive management updates from across the managed overlay, run policies over these updates and enforce actions where required.
8. **Management Plane:** The management plane has been introduced as an umbrella term for the trigger broker and policy server responsible for the operation of the managed overlay.

In the next chapter of this thesis the design outlined in this chapter is discussed in terms of its implementation. This is followed by a full evaluation of KBNCluster in comparison to KBNImpl, concluding with a discussion of the question this research is addressing, objectives, contribution and future work.

5 IMPLEMENTATION

In the previous chapter the design of KBNCluster has been discussed in terms of the decisions made in producing a prototype system to evaluate the research objectives of this thesis. In this chapter the implementation of the designed system is discussed in detail.

5.1 Introduction

This chapter introduces a prototype implementation of KBNCluster, comprising three components: the trigger broker, policy server and managed overlay, all implemented using the Java programming language (v 1.6.0.) As has been discussed in the design chapter, the trigger broker is implemented as an unmodified KBNImpl broker, whereas the policy server acts as both a KBNImpl publishing and subscribing client and incorporates a policy rule engine. Each of the individual components combine to implement a prototype version of KBNCluster, as described in the previous chapter. This was implemented in five main parts:

1. Extension of the KBNImpl to support the dynamic repositioning of publishing and subscribing clients, across the network.
2. The Medoid calculation implementation for both publisher and subscriber.
3. Use of a standalone KBNImpl broker to form and operate as the trigger broker an intermediary between the managed overlay (brokers & nodes) and policy server.
4. Implementation of a policy engine responsible for managing the clustering of clients around brokers of interest.
5. Integration of the logic for clustering and managing clients into a KBNImpl client subsequently forming the policy server.

Both the trigger broker and policy server used KBNImpl technology in KBNCluster, so the communication mechanisms present in KBNImpl could be utilised in the process of cluster management and inter-component communication. Therefore, rather than re-engineering a collection of TCP/IP communication agents and integrating these agents into each of the components, the KBNImpl itself, is utilised for messaging between all parts of the management plane.

5.2 Technology Selection

A number of technologies have been used in the implementation of KBNCluster which consist of the managed overlay (i.e. the network of KBNImpl brokers and clients), trigger broker and policy server combined. This section introduces those technologies, the communication mechanism used between components and the development tools used in the implementation of KBNCluster.

5.2.1 Core Technologies:

- **The Siena CBN (v 1.21)** [2] was extended by Keeney et. al. [6] to form the KBNImpl. The KBNImpl codebase was further extended through the design previously introduced forming KBNCluster, evaluated in this thesis.
- **Drools (v 4.0.7)** is the rule engine used in the policy server. Drools [49] was selected as it offers both an interface to write and edit rules as well as a rule engine to execute rules against data objects.
- **Jena (v 2.5.4)** [50] has been used extensively in KBNCluster as a way to access the knowledge represented in ontologies through its Java API.
- **Pellet (v 1.5.1)** [51] is a Java based OWL DL reasoner which is used in conjunction with Jena for the classification, inference, consistency checking and validation of ontological models.

5.2.2 Development Tools:

The following Integrated Development Environments (IDEs) have been used in the implementation of KBNCluster, primarily for the development and editing of ontologies, coding of applications, and design and debugging of policy rules. Each is discussed in turn:

1. **Protégé (v 3.3.1)** [52] Protégé was used for the development, editing, testing and reasoning (using Pellet [51]) of each of the ontological models used in the trigger broker, policy server and managed overlay.
2. **Eclipse (v 3.4.2)** [53] is primarily a Java IDE which has been used for the management, development and deployment of KBNCluster. Eclipse being an open-source IDE for multi-language development allows for plug-ins to be developed on top of the Eclipse IDE.
3. The **Drools workbench (v 4.0.7)** [49], is an example of such a plugin built on-top of the Eclipse IDE, allowing users to create a Drools project with the correct dependencies, as well as providing a specific GUI for editing and debugging rules. The Drools rule workbench has been used for all policy rule engineering.

Protégé, Eclipse and the Drools workbench have combined to provide the foundation for KBNCluster development and deployment, allowing OWL ontologies, Drools rules and the Java code base of KBNCluster to be edited and de-bugged.

5.2.3 Messaging Mechanisms:

Across KBNCluster, the KBNImpl communication mechanism is used wherever possible. In this approach there are two main types of messages used between brokers, publishers, subscribers, the trigger broker and the policy server. Both these message types are a single point of failure should the delivery mechanism or clients become unreachable.

Message Type 1: KBNImpl publications and subscriptions: The trigger broker, offering the functionality of a KBNImpl broker, routes incoming publications towards interested subscribers. The policy server issues expressions of interest for future management messages. These are satisfied through publications sourced from across the managed overlay via the trigger broker forming the backbone for the routing of management information between the managed overlay (brokers, publishers and subscribers) and the policy server.

Message Type 2: Direct KBNImpl to KBNImpl Communication: Brokers in the network can communicate with one another directly, and the policy server can communicate with brokers, using the brokers' receiver address in the form of *tcp:HostName:Port*. In KBNCluster this mechanism is used to direct clustering instructions from the policy server back to the brokers of the network. This was utilised as it allows for a single configuration message to contain multiple enforceable actions, in multiple cases, sent directly from the policy server to brokers in the network.

KBNCluster uses two types of messages, as introduced above. Publishers exist detached from any broker, they publish and never again re-connect to the broker. For this reason the publishers in the overlay communicate with the policy server using specially designed publications and subscriptions delivered and published addressed to UUID via the trigger broker.

In contrast to publishers subscribers' exist connected to their broker. A subscriber cannot exist without being connected to a broker. This makes communication with a subscriber, via their broker, as outlined in Message Type 2. There was no requirement, in KBNCluster, to implement direct communication from the policy server to the subscriber as the subscribers broker exists as an intermediary between both. It would however be possible for subscribers to communicate with the policy server through the use of a UUID, this feature is not however implemented in KBNCluster at this time.

5.3 Order of Operation

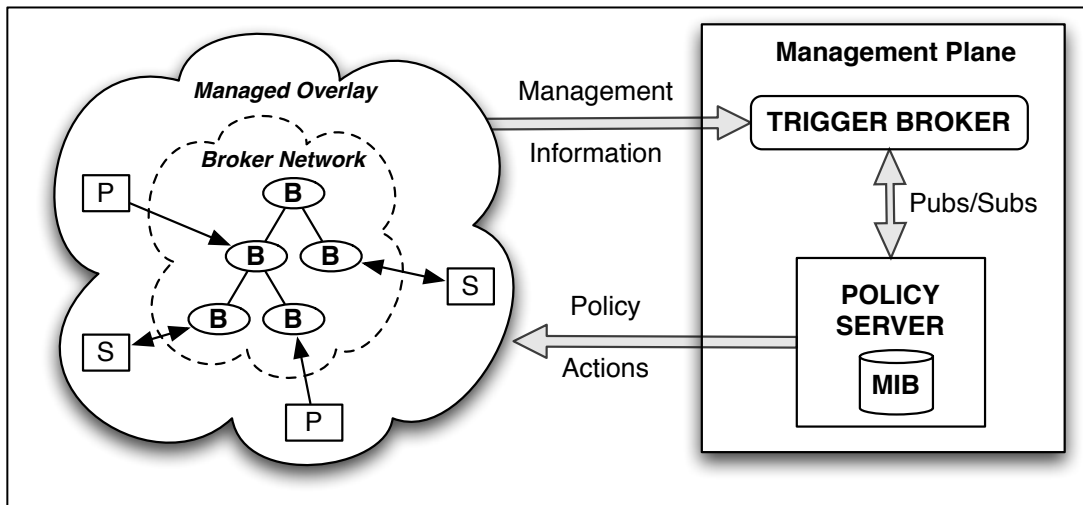


Figure 12: Components of KBNCluster

Shown in Figure 12 is the high-level component architecture, as has been introduced and discussed in the Design Chapter, 4, Section 4.2.. The evaluated KBNCluster is not a real-world deployment but an evaluation test-bed therefore the order of operation is strictly defined. The trigger broker must be started first to receive subscriptions from the policy server before nodes in the managed overlay begin to publish management data, being delivered across the Trigger Broker to the Policy Server.

1. Trigger Broker Start-up Process:

- a. Start a KBNImpl broker running on a user-defined port.
- b. Load the management ontology as discussed in the Design Chapter 4, Section 4.4, used to reason incoming messages.
- c. Await incoming subscriptions from the policy server, which are processed, organised and stored in the trigger broker's subscription tree.
- d. Wait for incoming publications, from nodes in the managed overlay containing management information. If any publications match stored subscriptions, route as notifications towards subscribers, such as the policy server.

2. Policy Server Start-up Process: (Can only be started after the trigger broker)

- a. Cluster the content ontology used within the managed overlay for routing messages from publisher to subscriber. This is the ontology used by KBNImpl, used for creating semantic publications and subscriptions routed from producer to consumer by message broker.
- b. Overlay the clustered ontology onto the list of available brokers, dividing the number of clusters across the set of available brokers.
- c. Subscribe to the trigger broker with subscriptions for future management data.

- d. Start timer-based policies specifying certain tasks to be performed at defined intervals.
 - e. Await management updates to arrive from the trigger broker.
- 3. Managed Overlay: (*Can only be started after the Trigger Broker and Policy Server*)**
- a. Managed Brokers (started first, before publishers or subscribers):**
 - i. Load the content ontology used for reasoning about messages.
 - ii. Start a KBNImpl broker using the server's host name.
 - iii. Publish to the trigger broker a semantic broker information message.
 - b. Managed Subscribing clients (Can be started once brokers have loaded):**
 - i. Creates a KBNImpl subscribing client.
 - ii. The managed subscribing client subscribes to an un-constrained choice of broker.
 - iii. The broker forwards to the trigger broker a subscriber information message, as discussed in the Design Chapter 4, Section 4.5.1.
 - iv. The policy server, upon receiving the subscriber's information message, calculates the correct cluster for the subscriber and sends this re-cluster instruction back to the broker to which the subscriber is attached in an XML configuration message. This message contains the identifier of the new broker to which the subscriber must reattach.
 - v. The subscriber un-subscribes from their old broker and re-subscribes, as instructed to do so by the policy server.
 - c. Managed Publishing clients (Can be started once brokers have loaded):**
 - i. The application initialises a KBNImpl publishing client.
 - ii. The publisher client subscribes to the trigger broker for all messages containing their UUID.
 - iii. The publisher client calculates their Medoid from the publications they wishes to publish and sends this to the trigger broker as a publisher info message, discussed in the Design Chapter 4, Section 4.5.1.
 - iv. The publisher client await a re-clustering instruction from the policy server and a decision as to which cluster they should join delivered to them as a notification, addressed to their UUID from the policy server, via the trigger broker.
 - v. The client publishes its publications to the broker and cluster, as suggested by the policy server.

The above order of operation is extended upon, in the next section, with implementation details relating to the detailed communication between the components of KBNCluster represented as sequence diagrams.

5.4 Communication Flow

This section outlines the communication between the managed overlay (brokers, publishers and subscribers) trigger broker and policy server. Sequence diagrams are used for modelling this communication and a modified form of UML sequence diagrams [54].

In these examples each message in the sequence diagrams is numbered, followed by a numbered list of descriptions, where each numbered item on the sequence diagram corresponds to the description provided in the list. The following communication sequences are presented and discussed in detail:

1. **All components** using publish/subscribe technology for communication.
2. Communications between the **broker**, trigger broker and policy server.
3. Communications between **publisher**, trigger broker and policy server.
4. Communications between **subscriber**, broker, trigger broker and policy server.

Once the above have been discussed, this section concludes with a review and summary of the various types of messages sent across KBNCluster, before discussing the implementation of the trigger broker, policy server and clustering process in turn, concluding with examples of starting a KBNCluster deployment.

5.4.1 Overall Communication

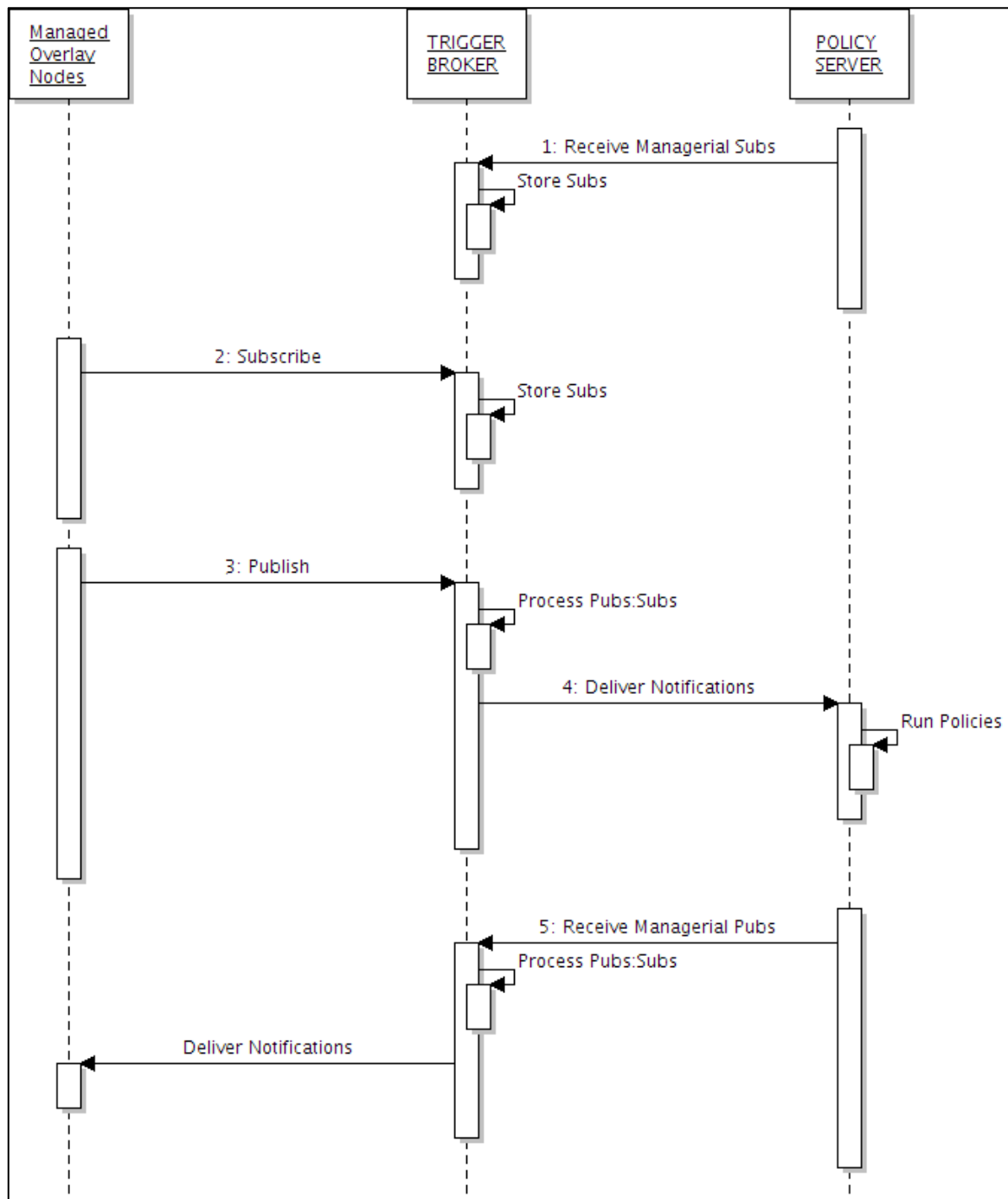


Figure 13: Overall Communication Mechanisms

(Managed overly nodes can be brokers, publishers or subscribers.)

1. The policy server registers an interests in management information through subscriptions to the trigger broker, stored in the trigger broker's subscription table.
2. The nodes (publishers) in the managed overlay register an interest in future publications containing their UUID. These are also sorted and stored in the trigger broker's subscription table.

3. Nodes in the managed overlay begin to publish management information to the trigger broker and these publications are checked against the trigger broker's subscription table.
4. If any of the subscriptions stored on the trigger broker match incoming publications, these are forwarded to matching subscribers, in this case the policy server, as notifications. These are used by the policy server to execute policy rules against the most up-to-date Managed Objects (MO) held in the Management Information Base (MIB).
5. If a policy fires, instructions from the policy directives are sent to the nodes of the managed overlay, using one of the communication mechanisms outlined in Section 5.2.3, either KBNImpl publish/subscribe or direct KBNImpl configuration messages.

This sequence shows the policy server receiving management updates from publishers and brokers across the managed overlay and communicating with the publishers, addressed using their UUID, using messages sent via, and matched by, the trigger broker.

5.4.2 Broker Focused Communication

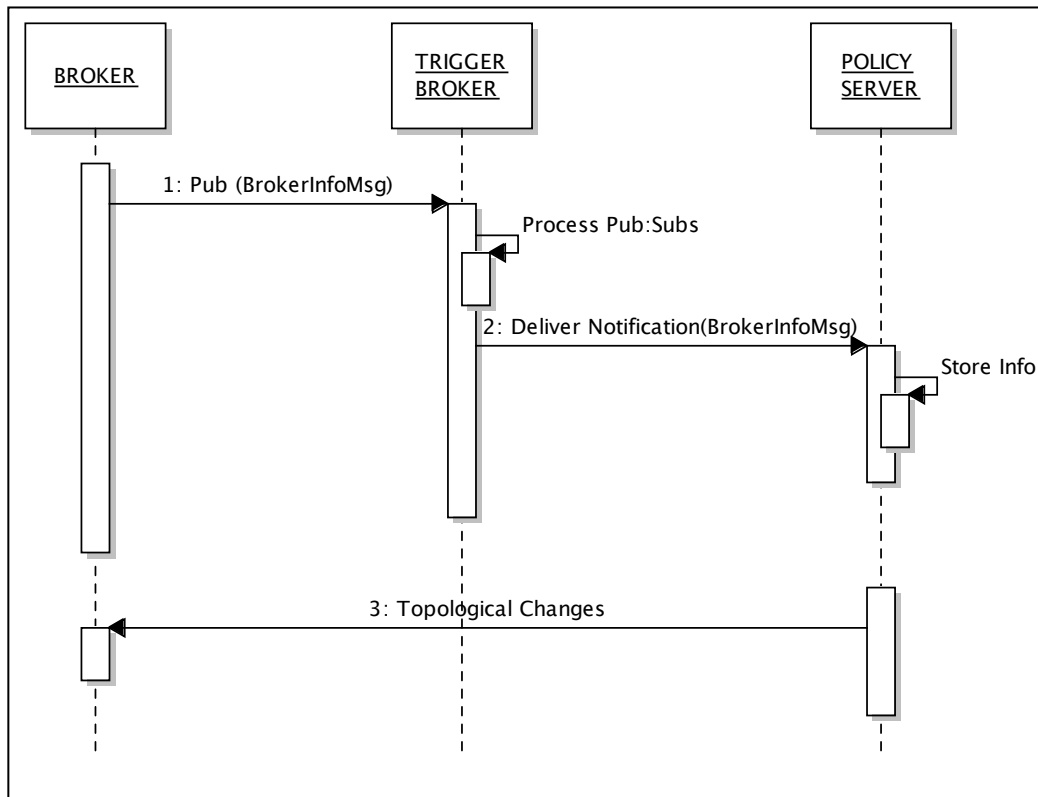


Figure 14: Broker Based Communication

1. The broker publishes to the trigger broker a broker information publication when requested (at a defined interval in policy) to do so by the policy server.
2. If a match is found for the broker information message against stored subscriptions, the trigger broker forwards this information message towards subscribers. When the policy server receives these management messages they are stored in the managed object (MO) representing that broker.
3. Whereas the communications represented through messages 1 and 2 are KBNImpl publications and subscriptions, message 3 is a direct configuration message sent from the policy server to the KBNImpl broker in question, allowing the policy server to issue commands directly to brokers when fired by policies. This is used as the broker is already monitoring a socket for incoming publications, subscriptions or configuration messages. The policy server can therefore direct a message directly to the broker, completely bypassing the trigger-broker.

The communication sequence presented above shows broker information messages being delivered to interested subscribers, which in this case is only the policy server. Once processed and if a policy fires, KBNCluster allows for topological changes/directives to be sent back to brokers, by the policy server, and the subscribers attached to the broker.

5.4.3 Publisher Focused Communication

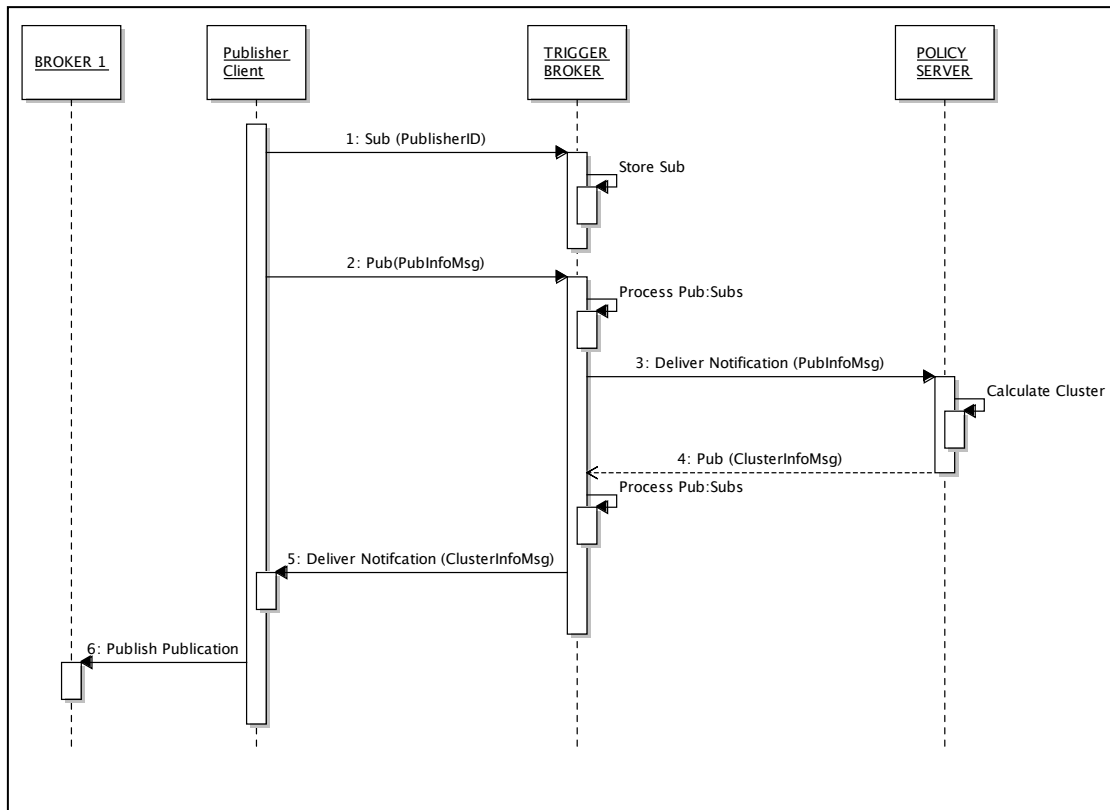


Figure 15: Publisher based communication

1. Publishers subscribe to the trigger broker using their UUID as an attribute in their subscription. The trigger broker stores this subscription allowing for any future publications arriving at the trigger broker, containing the publisher's UUID, to be forwarded to the publisher, as a notification.
2. The publisher, once subscribed, publishes to the trigger broker a publisher information message, as discussed in the Design Chapter 4, Section 4.5.1. The trigger broker forwards this publisher information messages to the policy server.
3. The trigger broker delivers publications, matching stored subscriptions to the policy server, as notifications. The policy server, upon receiving these notifications, updates the MO identified by the UUID of the publisher and executes any policies, including those responsible for calculating the suggested semantic cluster of the publisher against the MO.
4. Once a cluster has been identified for the publisher, the policy server sends to the trigger broker a publication containing the UUID of the publisher, identifying their most suitable suggested cluster.
5. The trigger broker identifies a match of the policy server's publication and the previously registered subscription from the publisher to their UUID. This publication contains clustering instruction for the publisher and is delivered to the publisher using their UUID element.

6. In the final step the publisher connects to the appropriate cluster, as calculated at the policy server, and publishes the publication towards the broker holding interested, clustered, subscribers.

The communication sequence above outlines the messages sent in the process of clustering publishers, who communicate with the policy server, via the trigger broker, finally publishing to a broker / cluster suggested as suiting their interests.

5.4.4 Subscriber Focused Communication

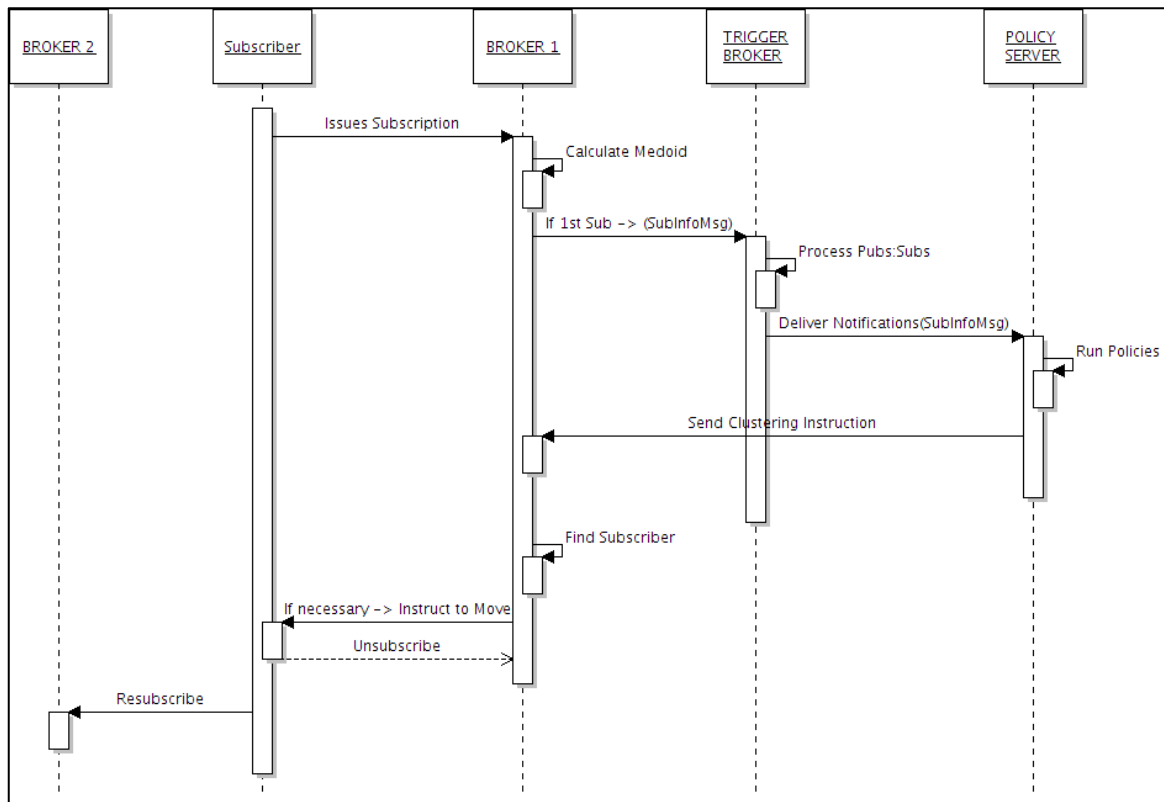


Figure 16: Subscriber based Communication

1. The subscriber initially issues their subscription to an un-constrained choice of broker across the network. A broker, upon receiving the first subscription of a subscriber, stores this subscription in their local subscription tree and calculates the subscriber's Medoid.
2. Once the Medoid of a subscriber has been calculated, the broker forwards this to the trigger broker as a subscriber information message, as discussed in the Design Chapter 4, Section 4.5.1. When the trigger broker receives this message it searches for matches against the incoming publication and stored subscriptions.
3. If a match to the incoming publication is found, the trigger broker forwards the publication to interested subscribers, in this case the policy server, as a notification. Upon receiving a notification the policy server updates the relevant MO and policy rules that implement the clustering algorithm are executed against it.
4. When the policy server has identified the correct cluster for the subscriber, this decision is communicated directly to the subscriber's broker. Rather than going through the trigger broker, the policy server in this case sends a KBNImpl configuration message directly to the broker, using its open socket.
5. Once the broker receives an instruction to move a subscriber, the subscriber is informed.
6. The subscriber un-subscribes from the broker they were attached to and re-connects to the cluster identified as matching their interests. There are not however re-clustered.

The communication sequence above shows messages being sent in the process of clustering subscribers from an initial and single subscription to a broker through to their re-clustering into an suggested cluster of interest.

5.4.5 Conclusions – Communication Flow

In this section the two types of messages, sent between the components of the managed overlay and policy server have been discussed. These are the only types of messages used in the management of KBNCluster (KBNImpl publications to subscriptions matching) and (direct socket-to-socket KBNImpl messages.)

The sequence diagrams presented in this section outline the high-level messaging used between all components in KBNCluster. All **KBNImpl messaging** occurs via the trigger broker whereas **direct messaging only** occurs between brokers and the policy server, using the KBNImpl brokers receiver addresses and open socket. These messages are not publications or subscriptions but highly configurable XML based messages, allowing an extendable number of managerial instructions to be delivered to brokers in the managed overlay, as part of KBNCluster. The communication relationships between all the components are summarised below:

- Publishers send to the policy server using **KBNImpl Messaging**.
- Subscribers send to the policy server, via their broker, using **KBNImpl Messaging**.
- Brokers send to the policy server using **KBNImpl Messaging**.
- Policy Server send to publishers using **KBNImpl Messaging**.
- Policy Server send to subscribers via their broker using **Direct Messaging**.
- Policy Server send to broker via **Direct Messaging**.

Subscribers in the first instance make an arbitrary decision as to the broker to join and subscribe to. They are then re-clustered using **Direct Messaging** between the policy server and the broker to which they first attached. All communication with subscribers is conducted using **Direct Messaging** between the policy server and the subscribers attached broker.

Publishers subscribe to the trigger broker for messages addressed to their UUID. Publishers always go to the trigger broker for clustering instructions; UUID's connect publisher and policy server, via the trigger broker. All of this communication occurs using **KBNImpl Messaging**.

In conclusion the two types of messaging KBNImpl-based and direct socket-to-socket messaging are used in KBNCluster. This section has introduced sequence diagrams for all communication between all components, and detailed the type of message used at each point. The subsequent sections of this chapter examine the trigger broker, policy server and clustering implementation in detail.

5.5 Trigger Broker

As discussed, the trigger broker is a standard KBNImpl broker, routing messages from the managed overlay to the policy server and management instructions back to publishers. This section discusses starting the trigger broker, its main classes of operation, and concludes with a summary of its key characteristics. The Trigger Broker was not implemented as part of this thesis it as “off the shelf” KBN broker used in KBNCluster.

5.5.1 Trigger Broker – Start-up Configuration

The trigger broker operates as a single KBNImpl broker, with no master or children. However it could, in future deployments, form part of a larger collection of brokers in a hierarchical tree, as required for reliability, itself clustered. In KBNCluster evaluated as part of this thesis, only a single trigger broker is required. This section discusses the configuration variables provided to the trigger broker in its configuration.

```
#!/bin/sh
java siena.StartServer -port <port> -tcp -log - -err -
```

Code Example 8: Starting the Trigger Broker

At the highest level the script, shown in Code Example 8 is the Linux Bash Script [55] that once executed, completes the following tasks, in order:

1. Loads the management message ontology identified in the *ontologies.conf* file. This holds a list of the ontologies that the trigger broker should load and subsequently route semantic messages. The trigger broker, in KBNCluster, only uses this single ontology, as discussed in the Design Chapter 4, Section 4.4.
2. Once the ontology has been loaded the trigger broker starts an un-clustered KBNImpl broker, on the port specified and the host IP address forming a *tcp:HostName:Port* address.
3. Once started the trigger broker awaits incoming subscriptions, from the policy server, storing these in its local subscription table and routing incoming publications to interested subscribers, where and if a match is made.

5.5.2 Trigger Broker – Main Classes

There are three main classes that are important to note in the trigger brokers operation. For each of these three classes the main operations of each are outlined. Each of these features were present in the KBNImpl used for the trigger broker and were not implemented as part of this thesis.

1. Starting a server – *siena.StartServer()* – Takes input configuration parameters and instantiates a broker using the following:

- a. The port on which the broker will run, **<port>** in Code Example 8,
 - b. Any master broker (n/a in the case of KBNCluster where a single trigger broker is being used as opposed to a hierarchical network of trigger brokers.)
 - c. Creates an object representing the Siena-hierarchical dispatcher of the broker populated with the configuration parameters taken from the start-up script. The dispatcher is responsible for routing incoming messages, storing subscriptions and notifying clients of matching publications to stored subscriptions.
 - d. Once populated with the configuration variables the dispatcher is started as a separate thread operating with the purpose of listening and processing incoming messages.
2. Message processing - ***siena.HierarchicalDispatcher()***
- a. The dispatcher listens on an open port for incoming messages. As they are received they are passed to the ***ProcessRequest*** method, which is responsible for determining the course of action to take for each message type, whether that be a publication, subscription or notification. The messages type is used to action an event, which can be one of the following:
 - 1) ***Publish()*** :
 - a) Searches the subscription tree of the broker for subscriptions matching the publication.
 - b) If a match exists it forwards the publication to any local or remote subscribers.
 - 2) ***Subscribe()*** :
 - a) Firstly the broker searches their set of subscriber objects, where each object represents a separate subscriber, to see if an entry already exists.
 - b) Either updates or creates an entry for the subscriber with the new subscription information contained in the subscription.
3. Ontology loading – ***extSiena.Ontology.OntSienaConfig()***
- a. Load all the ontologies stored in ***ontologies.conf***.
 - b. For each ontology loads the classes, properties and individuals.
 - c. Once loaded, the trigger broker can query a stored ***Attribute-Constraint*** against an incoming ***Attribute-Value***.

The three classes of the trigger broker provide a delivery mechanism for incoming publications to be matched against stored subscriptions and delivered as notifications if necessary. This uses the KBNImpl message-matching algorithm in the delivery of management information from the managed overlay to the policy server and subsequently for instructions to be sent back to publishers.

5.5.3 Trigger Broker - Summary of Key Characteristics

The key tasks performed by the trigger broker are summarised below:

- Utilises the management ontology (see Design Chapter 4, Section 4.4) to semantically match incoming publications to stored subscriptions, being delivered as notifications.
- Receives semantic subscriptions from the policy server and managed overlay publishers,
- Receives semantic publications from the policy server and managed overlay.
- Routes notifications from publishers, managed overlay **or** policy server, to the policy server **or** publishers where a match is made.

Having discussed the trigger broker in terms of start-up configuration, main classes, and providing a summary of key characteristics, the same information is presented for the policy server before discussing in detail the process of clustering. This chapter then concludes with an overview of starting each of the components of KBNCluster.

5.6 Policy Server

This section looks at the operation of the policy server responsible for both receiving management information from the managed overlay, storing this information in the MIB, as MOs, before executing affected policies against these stored MOs.

As with the trigger broker, the policy server uses direct socket-to-socket communication and publish/subscribe messages in communication between brokers, subscribers and publishers. The policy server incorporates KBNImpl client functionality, providing support for MO creation/updating, policy execution and event actions.

All policies used in either the development or evaluation of KBNCluster are included in Appendix C of this thesis. This section discusses the start-up sequence and configuration of the policy server and some of its main operational classes before concluding with an overview of its implementation.

5.6.1 Policy Server - Start-up Configuration

Shown in Code Example 9 is the Linux Bash Script [55] used in starting the policy server. A single value passed to the policy server is the address of the, formatted as a KBNImpl receiver address: *tcp:HostName:Port*.

```
#!/bin/sh
java ie.tcd.cs.kdeg.mecon.Cluster_Mgmt.Cluster_Scheduler
'tcp:134.226.38.135:50000'
```

Code Example 9: Starting the Policy Server

At the highest level the code, shown in Code Example 9, completes the following tasks in order:

1. Loads the management ontology from the *ontologies.conf* file used in semantic communication between the trigger broker, policy server and the managed overlay nodes.
2. Subscribes to the trigger broker with subscriptions for future management interests.
3. Loads and clusters the content ontology, used in the managed overlay for matching publications and subscriptions.
4. Clusters and overlays the content ontology onto the list of available brokers, stored in *brokers.txt* listing the brokers available for clustering.
5. Start the timer based policies designed to fire at defined intervals.
6. Await incoming MO updates, from the managed overlay via the trigger broker, storing these as they are received.

When a complete MO has been updated or created, the policy server then executes the appropriate set of rules over the MIB, firing actions as required. When fired, these actions enforce the change outlined in the policy.

5.6.2 Main Classes

This section looks at the main classes of the policy server and their use in the process of clustering. Presented are the key functions of KBNImpl publish/subscribe communication, MIB creation / updates and the firing of policy rules:

- a. **Subscriber – *drools.KBNSubscriber()***
 - i. Create a new notification object. This objects allows the trigger broker to notify the policy server of management information matches.
 - ii. Subscribe via the trigger broker for management messages, using the previously created notification object for message delivery.
 - iii. Sleep while awaiting incoming notifications, which are delivered through the notification object. When received, these are passed to the notification processor responsible for extracting the type of the notification, and the source of the notification be it broker, publisher or subscriber. Once the type of message has been determined, the message is passed to the MIB updater.
 - b. **Publisher – *drools.rules.actioner()***
 - i. Create a new, blank, publication object.
 - ii. Add publication elements to the object, this is the publication that without clustering would normally be published directly to a broker. In KBNCluster this publication(s) is used to advertise the publishers Medoid to the policy server and for clustering to occur.
 - iii. Generates and adds its UUID to the object.
 - iv. Send this publication to the trigger broker, delivering this to interested subscribers with management instructions.
2. **MIB Updater – *cluster_Mgnt.Updater()***
- i. Create an instance of an actioner object passed to the policy engine allowing actions to be instigated, as required, across the managed overlay.
 - ii. Provide methods to create or update a MO.
 - iii. Provide methods to get and set variables stored in the MOs.
 - iv. When a MO has either been created or updated, the complete MIB is passed to the policy server and the appropriate set of rules executed against each MO. If a policy rule fires, the policy server uses the actioner object previously created in step i to instigate a change within the managed overlay, as outlined in the policy.

5.6.3 MIB/MO Implementation

As has been discussed, each component in the managed overlay is either a broker, publisher or subscriber, and therefore there are three matching types of managed objects (MO) available to the policy server.

As the policy server receives a new management message, the MO representing that management node (broker, publisher or subscriber) is updated, or in the first instance, created. This is then used by the policy server to execute rules against the data the MOs hold.

Broker/subscriber/publisher information as discussed in the Design Chapter 4, Section 4.5.1 are used to update or create entries in the MIB. Shown in Figure 17 are UML class diagrams for broker, subscriber and publisher MO's.



Figure 17: UML Class Diagrams for Broker, Publisher and Subscriber

Each MO and indeed the MIB itself is extensible, in that the number of variables in each can be extended or reduced as required. The key to the MIB is its purpose is to act as a buffer between incoming management messages and the policy server. If the requirements of the MIB change, this can be represented by the addition or removal of elements in each MO.

5.6.4 Clustering Process

This section outlines the implementation of the process of clustering an OWL ontology and the overlapping of the ontology onto a collection of brokers. This is a two-stage process that first involves the partitioning of the ontology into clusters. The second stage involves the calculation of the cluster best suiting a client Medoid. When combined, this provides the clustering algorithm used and evaluated in KBNCluster, in the subsequent Evaluation Chapter.

5.6.4.1 Cluster Partitioning

Ontology partitioning requires no knowledge of the semantic meaning of the ontological elements labels, is language independent and is implemented using the following approach:

1. Load the input content ontology into a Jena ontological model [50]. This is the ontology used by the brokers, publishers and subscribers across the managed overlay.
2. Reason over the ontology, using Pellet [51].
3. Take each of the reasoned ontological top level root classes, their sub-classes and their instances and create a cluster.
4. Take each sub-class of the root classes, and if it has children, create a cluster for that class, all its sub-classes and instances. (For each bottom level class, Jena is queried to return the instances for that class.)
5. Follow the same process for each class in the ontological taxonomy, from the top level to bottom level class, creating parent-child clusters of classes and instances where such a relationship exists.

The result of the process is a hash map where the key to each entry is the cluster identifier and the values associated with each key are the ontological classes or instances, from the source ontology associated with the cluster.

The clustering process does not explicitly deal with ontological properties in the clustering process, which can be either object-properties (a relationship between instances) or data-type, which relate data values to instances. In KBNCluster, a client Medoid will never be returned as a property, but an instance, or class. Therefore there are no clusters created for properties, but for the classes and instances associated with those interlinking properties. Having created the clusters of content, the next section of this chapter looks at overlaying this clustered ontology onto a collection of brokers, assigning to each broker a cluster.

5.6.4.2 Overlaying an Ontology onto Brokers

As previously discussed, KBNCluster is constructed from a hierarchical managed overlay. Brokers are related to other brokers via a parent-child relationship. Having partitioned the ontology and created multiple semantic clusters, the next step in the process of clustering is to overlay the ontology onto the brokers of the hierarchical managed overlay. This is implemented using the following approach:

1. Load list of broker addresses from a configuration file from top to bottom of hierarchy.
2. Load list of ontological clusters. These clusters are represented by a key, the top level ontological element (parent) and a number of values associated with the key, these being the children of the top level element.
3. Divide the number of clusters by the number of brokers to calculate both the quotient and remainder of the division. This provides the clustering process with the number of clusters that can be assigned to each of the available brokers and those left over.
4. Loop through, from the top to the bottom of the hierarchy, assigning n clusters to each broker where n equals the quotient of the division. Once complete assign the remainder n' clusters to the top level brokers. In such an approach each broker, across the overlay, is assigned an equal number of clusters from the top-down. If there are any remaining clusters these are assigned to brokers at the top of the broker network.

KBNCluster operates on the principle of clustering the ontology and applying clusters to the broker hierarchy as they are formed in the ontology. The process loosely overlays the ontology over the broker hierarchy, so the two (ontology and broker hierarchy) merge together. An ontology is simply a weighted graph of nodes and edges connected via semantic relationships, the broker network is again simply a collection of nodes related via parent-child relationships, edges. These similarities are used in overlaying the ontology across the broker network. However no guarantee can exist that the same number of brokers or ontological elements exist, so there can be no absolute overlaying of an ontology across a given number of brokers. For this reason KBNCluster aims to overlay one ontological cluster per broker, from top to bottom, left to right. If however more clusters exist than brokers the top level brokers are assigned these remaining clusters. These top level brokers are involved in less routing than those at the bottom of the hierarchy hence they can process the extra load placed required to process more than one cluster.

Part of the process of storing clusters in KBNCluster sees the least specific clusters being stored in the beginning of *clusters.txt* and the most specific, formed lower down in the ontology, being added to the end of the list. Similarly, *brokers.txt* holds the highest-level brokers at the top of the list and the lowest level brokers at the bottom. The clusters are then overlaid onto the broker network, with the most general interests being assigned to the top of broker hierarchy, and those clusters with the most specific interests being assigned to the bottom of the hierarchy.

5.6.4.3 Calculating Client Cluster Placement

Section 5.6.4.1 discussed creating clusters of content and in the previous Section 5.6.4.2 these clusters were overlaid onto the broker network. This process assigns to each broker a cluster of ontological concepts where every cluster is a collection of ontological URLs representing a key (i.e. a parent concept that acts as a top level cluster identifier) and the set of values, classes and instances from the ontology associated with that cluster (children).

It is possible to calculate which cluster a client should be placed within using the following algorithm, previously discussed in the Design Chapter 4, Section 4.3. Given, Medoid, an ontological URL:

1. Search through all the values in cluster hash map.
2. Retrieve the cluster ID of the cluster containing a matching value, the Medoid URL only ever residing in a single cluster, hence two clusters will never be suggested.
3. Return the broker responsible for the cluster identified by the cluster identifier (the key).

5.6.5 Summary of Key Characteristics

This section concludes with an overview of the key characteristics of the policy server which are summarised as:

- Operates as a publishing and subscribing KBNImpl client.
- Subscribes to the trigger broker with subscriptions for future management messages produced by the nodes (broker, publishers and subscribers) across the managed overlay.
- Publishes to the trigger broker with instructions for publishers.
- Communicates with brokers (and their subscribers indirectly), using socket-to-socket communication.
- Receives notifications from the trigger broker.
- Updates and maintains MOs for brokers, publishers and subscribers in the MIB.
- Runs policy rules against the MIB as MOs are created or updated running the relevant policies for the last type of update be it publisher, subscriber or broker.
- Clusters and overlays an ontology on a collection of brokers.
- Calculates and delivers instructions on the placement of clients into the previously created clusters.

The final section of this chapter looks at the process of starting brokers, publishers and subscribers before discussing conclusions and summarising the implementation presented.

5.7 Starting Brokers, Publisher and Subscriber

This section discusses starting the various components of the managed overlay once the trigger broker and policy server have been started, as documented in Section 5.5.1 and 5.6.1 respectively. The order in which the remaining components must be started is as follows: brokers first, followed by either subscribers or publishers. In the evaluation of KBNCluster brokers are started first, followed by subscribers and finally by publishers.

5.7.1 Starting a Broker

If the publish/subscribe network is hierarchical, then the first step in deployment is the configuration of the network hierarchy. This defines the number of brokers and the relationship between them, in the case of KBNCluster this is done using a master and child relationship, and can result in one of the following states:

- No master, no children, single broker - similar in operation to the trigger broker.
- A single master but no children - in effect the same as a single broker.
- No master but one or more children.
- Both a single master and one or more children.

Brokers are configured only to know their master: they have no indication of whether children are subscribing clients or child brokers exist.

The Bash script [55] shown in Code Example 10 configures and initiates the top level master broker, with no master itself, running on port *1500* with a trigger broker address of *tcp:134.226.38.135:50000*.

```
#!/bin/sh
java -classpath $CLASSPATH siena.StartServer -port 1500 -tcp -log - -
err - -TRIG_BROKER tcp:134.226.38.135:50000
```

Code Example 10: Starting the Top Level Master Broker

Code Example 11 shows a child broker using port *2000*. This broker looks to the broker addressed at *tcp:134.226.38.135:1500* as its master and to the trigger broker addressed at *tcp:134.226.38.135:50000*. Each broker when receiving a publication forwards this to its master, up across and back down the the tree. This allows any broker in the network to match any publication to stored subscriptions across the network. In this example the publication is forwarded to the brokers master: *tcp:134.226.38.135:1500*.

```
java -classpath $CLASSPATH siena.StartServer -port 2000
-master tcp:134.226.38.135:1500 -tcp -log - -err -
-TRIG_BROKER tcp:134.226.38.135:50000
```

Code Example 11: Starting a Sub Broker

5.7.2 Starting a Publisher

```
1  public void ExamplePublisher() {
2      Publisher pub = new Publisher();
3      Notification n = new Notification();
4      //Populate Notification with 1 or more (Name,Value,Type)
5      n.putAttribute("PubrInfoMsg",
5.1  "http://kdeg.cs.tcd.ie/KBNFaults.owl/#publisher\_Info");
6      n.putAttribute("publisherID", getUUID());
7      n.putAttribute("medoid", getMedoid());
8      n.putAttribute("pubTime", System.currentTimeMillis());
9      n.putAttribute("currentMaster", getCurrentMaster());
10     pub.addNotification(n);
11     pub.sendConfig("tcp:135.226.38.135:50000");
12     try {
13         Thread.sleep(50)
14     } catch (InterruptedException e) {
15         e.printStackTrace();
16     }
17     pub.startPublishing();
18 }
```

Code Example 12: Example Publishing Client

In comparison to brokers, publishers and subscriber are started using Java application, in code. Shown, in Code Example 12 is an example of starting a KBNImpl publishing client **where the code line numbers above correspond to the numbered bullets below:**

- 2: Create a new publisher object.
- 3: Create a new notification object, a publication.
- 4: Populate the notification object with **Attribute-Values**.
- 5/5.1: Add in semantic reference tagging the message as being a **publisher_Info** message.
- 6: Add in reference to publishers UUID using the **getUUID()** method.
- 7: Add in reference to publishers mediod using the **getMedoid()** method.
- 8: Add in publication time using the system time.
- 9: Add in the publishers current master, if any, using the **getCurrentMaster()** method.
- 10: Add the notification to the publisher object.
- 11: Send to the trigger broker a publisher configuration message containing the publishers UUID, Medoid, current master and publication time.

- 13: Sleep for 50ms, an arbitrary period, to allow the policy server's calculation as to the publishers suggested cluster and for these instructions to be sent back to the publisher. KBNCluster is not a production ready system hence this arbitrary time period is added between subscription and the next step publishing.
- 17: Start the publishing client addressed to the cluster suggested by the policy server.

5.7.3 Starting a Subscriber

In comparison to starting a publisher, subscribers require both an instance of a subscribing client and a notification object responsible for monitoring incoming notifications. Publishers publish their publication to a broker and then disconnect whereas subscribers leave a thread open for a broker to be able deliver notifications back to the subscriber, when a match occurs.

```
1 public void ExampleSubscriber() throws SienaException {
2     ThinClient subscriber = new ThinClient("tcp:134.226.38.135:2000");
3     SubscriberNotifier notify = new SubscriberNotifier();
4     Filter f = new Filter();
5     //Populate Filter with one or more (Name,Operator,Value,Type)
6     f.addConstraint("OntClass",ExtOp.MORESPEC,"http://conf#University");
7     f.addConstraint("Instance_ISA",ExtOp.ISA, http://conf#Ireland);
8     subscriber.subscribe(f, notify);
9 }
```

Code Example 13: Example Subscriber Client

Shown in Code Example 13 above is the code used by a subscriber to issue subscriptions to a broker. Again the line numbers above correspond to the numbered items below:

2. Create a thin client object that ties together the message filter and notification object into a single object responsible for communicating with a KBNImpl message broker.
3. Once the thin client object has been created, a notification object is constructed to provide a mechanism for publications to be delivered to a subscriber, as a notification.
4. A new blank subscription filter is created.
5. Populate the filter with **Attribute-Constraints**.
6. Add an ontological filter for all classes more specific than **#University**.
7. Add an ontological filter for all instances of the country **#Ireland**.
8. Subscribe to the chosen broker (an un-constrained choice) with the subscription filter and open a threaded listener object used for notifications to be delivered back from the broker network to the subscriber.

5.8 Conclusions and Summary of Technical Discussions

This chapter has outlined the implementation of KBNCluster, the types of messages sent, and provided an overview of key implementation code. The order of operation, for brokers, publishers/subscribers, the trigger broker and policy server have been outlined and the key characteristics of each element of KBNClusters implementation outlined. The managed overlay and management plane's implementation have been discussed and their interactions, in the process of clustering clients around common brokers of interest, as KBNCluster. In addition to this detailed communication flows have been outlined between:

1. Publisher and subscriber components.
2. Brokers, the trigger broker and policy server.
3. Publishers, the trigger broker and policy server.
4. Subscribers, brokers, the trigger broker and the policy server.

The two types of messages, KBNImpl publish/subscribe and direct socket-to-socket messaging have been discussed in detail, and their occurrence across the network discussed. The start-up characteristics of the trigger broker, policy server, message brokers, publishers and subscribers in either script or code have been presented.

For both the trigger broker and policy server, the key classes and characteristics of both have been discussed. The clustering process has been detailed with regards to the partitioning of an ontology and the ontology subsequently being overlaid onto the broker network. Where the main operating classes of KBNClusters implementation have been discussed, they have been looked at in terms of the requirements for clustering publishers and subscribers.

This implementation delivers the operational configuration of KBNCluster, described in the design chapter, that is used in the next chapter, evaluating the performance of KBNCluster in terms of the cost involved in moving clients around the network, storing management data, and the effect of being clustered on subscribers.

6 EVALUATION

The evaluation presented in this chapter is conducted upon a prototype implementation of KBNCluster designed to evaluate the efficiency of the approach taken to clustering. In this thesis two approaches to clustering are evaluated:

The first approach uses **Static clustering** in which brokers, publishers and subscribers are informed of the inter-relationships between one another before start-up. Therefore if this relationship changes, all the logic embedded in every client and broker must also change, through a full cold system re-start.

The second approach uses a **Managed Dynamic Clustering** approach that removes the necessity for embedding clustering instructions into the clients themselves. Publishers, subscribers and brokers are instructed by a management system, as to how to cluster. The key motivator for such a dynamic management approach is that changes in operational clustering policies can be made and enforced in KBNCluster, without requiring any change of the logic held in the brokers, publishers or subscribers.

The two approaches to clustering are introduced and evaluated across four sections:

- **Section 6.2 (Static Approach to Clustering)** – This section evaluates an initial experiment, conducted in the early stages of this research, into static clustering, motivating additional research to a managed dynamic approach to clustering, implemented as KBNCluster.
- **Section 6.3 (KBNImpl Operational Costs)** – This section evaluates operational costs associated with the use of an implementation of KBNImpl including semantic publication/subscription operator usage cost, subscription tree search times and the hop count metric, used in the evaluation of KBNCluster.
- **Section 6.4 (Costs Associated with Dynamic Clustering)** – This section evaluates some of the operational costs associated with applying a dynamic approach to KBNCluster. This includes management data collection, storage, policy execution and client mobility.
- **Section 6.5 (Dynamic Clustering Evaluation)** – This final section evaluates the overall approach of KBNCluster in terms of dynamically assigning publishers and subscribers to clusters of interests and the effect this has on the performance of the publish/subscribe paradigm. This section repeats the previous experiments conducted into a static approach to clustering, using the dynamic prototype implementation of KBNCluster.

The evaluation methodology taken works on the principle of multiple layered levels of evaluation, each building upon the last and being fully evaluated in a final experiment. In the first stage of evaluation a static approach to clustering is evaluated to establish whether clustering was of any benefit to an implementation of knowledge-based networking, KBNImpl. The second strand of evaluation evaluated KBNImpl itself in terms of the cost of introducing semantic operators and

types. The third strand of evaluation looked at the cost of storing and executing policies, collecting management data and client mobility. This presented an evaluation of the cost to manage, collect data and move clients. The final evaluation took the results of strands 1, 2 and 3 into account, fully evaluating a dynamic approach to clustering against a non-clustered topology. The evaluation was built on the following four principles:

- Establish whether clustering was feasible and effect in KBN implementations.
- Establish the cost of using semantics, as opposed to not using semantics.
- Evaluate management data collection, execution and mobility.
- **Evaluate a KBNCluster against KBNImpl in terms of performance gains.**

The evaluation approach outlined above provides a step-by-step evaluation of the separate components of a KBNImpl deployment, the effect management has upon such an implementation and finally an evaluation of the benefits introduced to KBNImpl when clustered as KBNCluster.

6.1 Experimental Setup

This section discusses, and outlines, some of the experimental set-up used across all sections of this evaluation, in terms of ontologies used, experimental platforms and the complete set of experimental metrics used across all experiments.

6.1.1 Ontologies

Through this evaluation chapter two ontologies were used. The first for Section 6.2 (Static Approach to Clustering) is discussed in Appendix A, included in electronic form in Appendix D “*subjectCat.owl*”. The second ontology, used in Sections 6.3, 6.4 and 6.5 “*confOf.owl*” (also included in Appendix D) is based around an academic conference, taken from the Ontology Alignment Evaluation Initiative [56] (OAEI2007).

The Static Approach to Clustering experiment (Section 6.2) used a different ontology than the remainder of experimentation as this ontology was deemed only satisfactory for use in a motivation initial case study, and lacked the range of instances, classes and properties required for fully evaluating KBNCluster. However all experimentation conducted into static clustering, in sections 6.3 and 6.4 are complimented by a full dynamic clustering experiment presented in 6.5. In section (6.2) an initial case-study is presented, in sections 6.3 and 6.4 experiments are conducted into static clustering using the same ontology as that used in section 6.5. Therefore this allows the dynamic experiments presented in sections 6.3 and 6.4 to be compared to the full and dynamic evaluation presented in section 6.5.

Thus the ontology used in experiments 6.3, 6.4 and 6.5, “*confOf.owl*” was selected and then extended to better match the median size of a representative set of OAEI2007 test ontologies, shown in Table 10. The “*confOf.owl*” ontology originally consisted of 23 classes, 23 data-type properties and 13 object-properties and was extended to consist of 45 classes that were populated with a total of 19 individuals. The existing 23 data-type and 13 object-properties left unchanged.

Table 10 presents the characteristics of five OAEI2007 ontologies, included in Appendix D, that were tested against the KBNCluster algorithm for cluster creation and placement accuracy. Each of the ontological elements (classes and instances) in each of the five test ontologies were passed to the clustering algorithm (Design Chapter 4, Section 4.3.5) to successfully test its consistent operation over different data sources. The results of this subjective experimentation is included for reference in Appendix B. This data, although interesting, can only be evaluated subjectively, in terms of how well a concept is placed into a specific cluster. The hop count metric, introduced later in this chapter, identifies objectively how well subscribers and publishers are clustered in terms of distance between one another and is deemed a direct reflection of the efficiency of the clustering algorithm.

Ontology Tested:	Classes:	Individuals:	Properties:
<i>confOf.owl</i>	45	19	36
<i>iswc.owl</i>	33	50	37
<i>ekaw.owl</i>	73	0	33
<i>swrc.owl</i>	55	0	74
<i>security.owl</i>	446	37	206

Table 10: Test Ontology Characteristics

6.1.2 Platforms

This experimentation has been conducted over two experimental platforms. The first distributed overlay platform utilised the PlanetLab [57] research network. This is a global overlay in which developers can deploy and test distributed networking applications. Experiment 6.2 was deployed to provide a geographically wide scale KBNImpl overlay. This experiment established the effect of clustering on a KBNImpl deployment, and the experience in network management of a statically clustered KBNImpl motivated future research into a dynamic approach to clustering. After conducting a full-scale deployment of static clustering, across the PlanetLab network, all other experiments were performed on a single host, like that used in evaluating KBNMap by Guo [4], for the following reasons:

1. **Synchronised Timing:** Using the PlanetLab architecture, or any distributed architecture, makes it very difficult to guarantee fine-grained synchronisation of the clocks in the various machines. This is especially true on PlanetLab nodes as administrator “root” access was not available to manipulate the system clocks, making distributed timing difficult.
2. **Resource ceilings:** At the time of deployment (2008) PlanetLab nodes were very limited in terms of the CPU and Memory resources, which limited the scope of experimentation.
3. **Node reliability:** In using PlanetLab it was found that there were many other researchers deploying experimental code onto the network, which, together with other operational server outages, would often cause unexpected node failure. In addition, well resourced nodes, would frequently become rapidly unusable as other researchers searched for more highly powered nodes for their experiments. Thus the reliability of nodes on PlanetLab was found to be less than adequate and as robustness to infrastructure failure was not a goal of the research, this was an impediment to future PlanetLab experimentation.

The machine used by Guo in [4] had the following specifications: Dell Inspiron 9300 laptop with 1.73 GHz Intel processor, 2GB of RAM, running Windows XP Service Pack 2. For Java-based tools, Sun’s JDK 1.6.0 was used. All tests were run at least 20 times to provide statistically appropriate averages.

The machine used in this thesis to evaluate KBNCluster had the following specifications: Dell Dimension 9200 Desktop PC with a 2.66 GHz Intel Core 2 Duo processor with 4GB of RAM running Ubuntu Linux 10.10. For Java-based tools, Sun’s JDK 1.6.0 was used. All tests were

either performed once when re-creatable each experimental run, i.e. the data is the same regardless of the number of runs. In other experiments sensitivity analysis is performed to establish statistically appropriate averages.

6.1.3 Metrics

In this section evaluation metrics are outlined for experiments where publish/subscribe performance is evaluated, and summarised for clarity. These metrics are derived from the work of Keeney et. al. in [58] which provides initial benchmarking characteristics of a KBNImpl deployment. With regard to KBNCluster this section aims to identify and motivate the commonalities and differences between data variables and the specific ranges of parameters chosen for each experiment. The variables are:

1. **No. of runs:** This refers to the number of times the experiment was run.
2. **No. of brokers:** This refers to the number of brokers used in the experiment. Where this value is greater than one, the brokers are arranged in a hierarchical topology.
3. **No. of subscribers:** This refers to the number of subscribers used in each experiment. Each subscriber is represented by a separate client with a number of subscription.
4. **No. of subscription filters per subscriber:** This refers to the number of subscription filters in each client's subscription.
5. **No. of subscription attributes:** Each subscription filter is made up of one or more (*Name, Type, Operator, Value*) *Attribute Constraints*. This value represents the number of these quadruples in each subscription filter.
6. **Subscription frequency:** This refers to the delay introduced between individual subscribers submitting their different subscriptions into the broker network.
7. **No. of publishers:** This refers to the number of unique publishers used in experiment. A separate client represents each publisher.
8. **No. of publications per publisher:** This value refers to the total number of publications each publisher client issues. Multiplying the number of publishers by the number of publications per client therefore provides the total number of publications used in each experiment.
9. **No. of publication attributes:** Each publication is made up from one or more (*Name, Type, Value*) *Attribute Value* pairs. This metric refers to the number of these triples per publication.
10. **Publication frequency:** This value represents the delay introduced between publications submitted to the broker network by individual publisher clients.
11. **No. of notifications delivered:** This value refers to the number of notifications delivered to a subscriber, where one or more of their subscriptions has been matched to a publication.

With regard to subscription and publication frequency, these values represent the time, in milliseconds (ms), between each publisher or subscriber submitting their publication or

subscription to a broker. For example if an experiment has 250 subscribers and a subscription frequency of 250ms then each unique subscriber subscribes with an interval of 250ms between each subscription. Shown in Table 11 is a breakdown of the range of metrics used in each experiment, introduced above, discussed in this section.

6.1.4 Sensitivity Analysis

For a number of experiments the data value presented does not suffer from any change, regardless of the number of times run the experiment is run. For example in the experiments regarding the number of hops taken to deliver a message, regardless of how many times the experiment is run the number of hops taken stays exactly the same.

There are some experiments, particularly those involving measuring time, where when run there is a slight variance in the measurement recorded. In addressing this variance in each of these experiments the data point is calculated for 1,3,5,10 and 15 runs. So in the first experiment (1 run) only 1 value is recorded, in the second experiment (3 runs) the mean from 3 experiments is recorded, the same for 5, 10 and 15 experimental runs.

The trend between each of the mean values (1, 3, 5, 10, 15 runs) is compared to establish whether with more experimental data points, the mean value changes i.e. whether as more experimental runs are added to the mean calculation is there any fluctuation in the data values recorded. From this sensitivity analysis the choice of using five mean experimental runs is justified.

Sensitivity analysis is presented in Appendix E for the following experiments:

- 6.3.2 Subscription Tree Search Time
- 6.4.2.4 Subscription Processing Times
- 6.4.2.5 Publication Processing Times
- 6.4.2.6 Pub-to-Sub Delivery Times
- 6.4.3.1 Moving Broker & Moving All Subscribers

In the experimental matrix presented on the next page where five runs are recorded the mean of five experimental runs has been shown to be stable as a measure of the experiment. Where one experimental run is recorded there is no fluctuation in the data points regardless of the number of times the experiment is run therefore only one value needs to be recorded.

<i>Experiments which use publish/subscribe messages:</i>	No of exp runs	No of brokers	No of subscribers	No of subscription filters per subscriber	No of subscription attributes	Sub freq	No of publishers	No of publications per publishers	No of publication attributes	Pub freq:	No of delivered notifications
Static Approach To Clustering											
Subscription Tree Size	1	37	75000	6	0-4	250	1500	1	15	250ms	N/A
KBNImp Operator Costs											
KBNImp Operator Usage	1	1	500-3000	2	1	250	250	2	1	250ms	300
Subscription Tree Search	5	1	1000-6000	2	4	250	250	2	4	250ms	300
Hop Count Measurement	1	11	3000	3000	4	250	3000	4	4	250ms	3000
Hop Count Timing	1	11	3000	3000	4	250	3000	4	4	250ms	3000
Data Collection Costs											
Subscription Processing	5	1	500	1	2-5	9ms	N/A	N/A	N/A	N/A	N/A
Publication Processing	5	1	500	1	2-5	9ms	120	1	2-5	250ms	35
Pub-to-Sub Delivery Times	5	1	500	1	2-5	9ms	120	1	2-5	250ms	35
Mobility Costs											
Moving Broker & Subscribers	5	2	200-1000	2	1-6	9ms	N/A	N/A	N/A	N/A	N/A
Moving All Subscribers	5	2	200-1000	2	1-6	9ms	N/A	N/A	N/A	N/A	N/A
Dynamic Clustering Evaluation											
Subscription Tree Size	1	11	250	2	1-6	9ms	N/A	N/A	N/A	N/A	N/A
Notifications Delivered	1	11	250	2	1-6	9ms	1000	1	1-5	250ms	98
Hop Count in Delivery	1	11	250	2	1-6	9ms	1000	1	1-5	250ms	98
Re-Clustering	1	11	250	2	1-6	9ms	1000	1	1-5	250ms	98

Table 11: Matrix of Experimental Metrics

In each experiment a number of decisions were made as to the upper and lower limits of data values, their exact value or range. Some were designed to find the upper limit of the data range and some designed so the experimenter could assure clustering decisions were being made correctly, manually interpreting experimental results. The specific experimental design decisions are presented in more detail in the relevant sections, but the following provides an outline of this chapter.

- **Static Approach to Clustering (See Section 6.2):** In this evaluation statically encoded rules guided clustering. For this reason sufficiently high levels of publishers and subscribers were used, across a large number of brokers in order to detect the impact of clustering. Operating with a large number of brokers offers practical insight into the practical difficulty of a static clustering management approach.
- **Data Storage and Policy Execution Costs (See Section 6.4.1):** This section is excluded due to this being an evaluation of storing managed objects (MOs) in the management information base (MIB) of the policy server. These values used in Section 6.4.1 were chosen to push the upper bounds of incoming MOs and the execution of policies against them, thereby stressing the performance of management system as a potential bottleneck, but do not fit into the publish/subscribe characteristics of the table.
- **Data Collection Costs (See Section 6.4.2):** When choosing this range of data values, a fixed number of publishers and subscribers were chosen to represent a high load for the number of brokers used in the evaluation of KBNCluster, a stress test.
- **Mobility Costs (See Section 6.4.3):** In this section, a range of subscribers were moved in each experiment to identify the variance in costs associated with each move. This was progressively repeated using larger numbers of subscribers. There were no publishers used in the experiment, as all that was being evaluated was a period of routing service disruption due to the movement of subscribers.
- **Dynamic Clustering Evaluation (See Section 6.5):** These values were chosen to be representative of 50% of the load used in the data collection costs experiments. However a much higher rate of publications to subscriptions was chosen, as the random creation of both resulted in fewer matches (notifications). Additionally the ratio of incoming publications to stored subscriptions was chosen to push the performance limit of the brokers in searching incoming publications against stored subscriptions, one of the more costly KBNImpl operations.

This section has introduced a matrix of experimental parameters for each experiment. Included in Appendix D, the data DVD, are the ontologies used in all experiments, and the publication and subscription data set that, when combined with this experimental matrix, supports the reproduction of the experiments of this chapter.

6.2 Static Approach to Clustering

6.2.1 Introduction

This first experiment was conducted as a test case of the effect of clustering on KBNImpl designed to show that clustering improved the performance of brokers by demonstrating that the subscription tree size is reduced on brokers across the network through clustering. The set of subscription and publication data used was designed to offer ample scope for occurrences of subscription covering. If subscription covering occurs in a broker this reduces the overall number of subscriptions that need to be matched by a broker to incoming publications and therefore increases routing efficiency in terms of the rate at which incoming publications can be matched to stored subscriptions. One subscription is more likely to cover another if they are about the same content or, more specifically, the same set of *Attribute-Constraints*. In particular, computationally expensive semantic subscriptions (which must invoke an ontology reasoner) are more likely to exhibit coverage if they are from the same area of the knowledge base (ontology). At a network wide level, a lower number of root subscriptions in the root node acts as a proxy metric for subscription covering occurrences, via clustering.

In this first experiment a single generic ontology was formed around a taxonomy of classes used in a one-cluster topology; this is then split, *manually*, into two sub-ontologies for a two-cluster topology and similarly, manually, into three sub-ontologies for a three-cluster topology. Each cluster in the broker topology then utilises each of the newer, smaller ontology strands for publishing, routing and subscribing content. The process of splitting a single ontology manually into multiple clusters is only used in this initial experiment as a baseline for the following experiments using automated clustering and the whole ontology.

6.2.2 Experimental Metrics

Shown in Table 12 are the experimental metrics used in all experiments conducted as part of this experiment. Please refer to Section 6.1.3 for detailed discussion behind each of these terms. For detailed discussion and examples of subscriptions and publications see Appendix D.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:			
1	37			
No of subscribers:	No of subscription filters per subscriber:	No of subscription attributes:	Sub delay:	
75000	6	0-4	250ms	
No of publishers:	No of publications per publisher:	No of publication attributes:	Pub delay:	No of notifications:
1500	1	15	250ms	Not recorded

Table 12: Experimental set-up metrics used in experiments

6.2.3 Experimental Setup

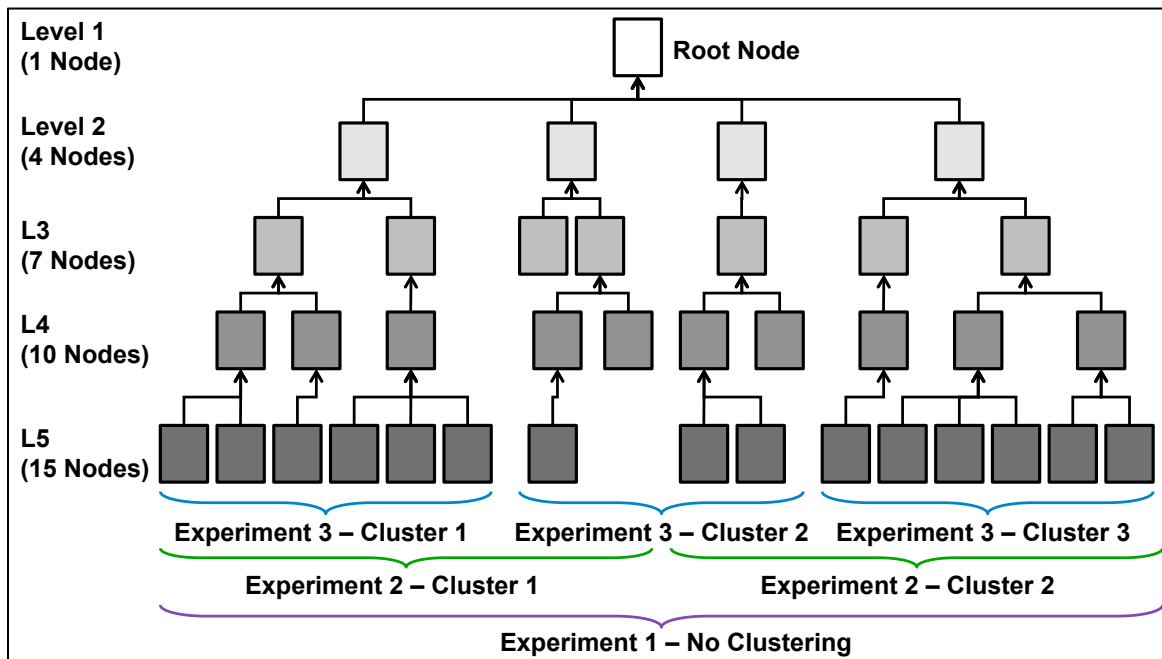


Figure 18: Broker Hierarchy

Shown in Figure 18 is the broker hierarchy used in this experiment, where each shaded square represents a single broker. Thirty-seven nodes were chosen and used, this being the upper limit of nodes that could be accurately monitored across the PlanetLab [57] research platform by a single researcher, in a static approach.

In each experiment, the ontology was partitioned and split into multiple physical sub ontologies and each sub-ontology applied to a section of the broker network as shown above. Publishers and subscribers then publish or subscribe from a selected sub-ontology into parts of the broker network allocated to that sub-ontology. This is implemented by clients submitting messages to certain brokers, based around common interests of the sub-ontology, as defined in a static spread sheet, which is parsed and used by a message scheduler in each client, across the network, Full details of this experimental set-up can be found in Appendix A of this thesis.

6.2.4 Results

Shown in Figure 19 are the root subscriptions held by the root broker in the no-cluster, 2-cluster and 3-cluster configurations, shown in Figure 18. Figure 20 presents the total unique subscriptions held by a typical broker at level three in the broker network. The data in these figures shows that clustering has reduced the total number of subscriptions, received by brokers as sampled at level three of the broker network. Due to the limited number of subscription combinations that span the network and the clusters within it, the number of root subscriptions eventually falls to a stable number, as new subscriptions, that are passed to the root, aggregate with existing root node

subscriptions. A key finding that can be drawn from Figure 19 is that with clustering subscription aggregation at the root node is greatly increased and covering converges much faster.

The data from the brokers shown in Figure 19 (master) and Figure 20 (level 3) are chosen as representative of key nodes in the network. The master broker must receive all root level subscriptions from its children (i.e. all children) and therefore truly represents the load of the overall network. Similarly the broker from level three can be seen as a representative node, responsible for a high number of its children's subscriptions.

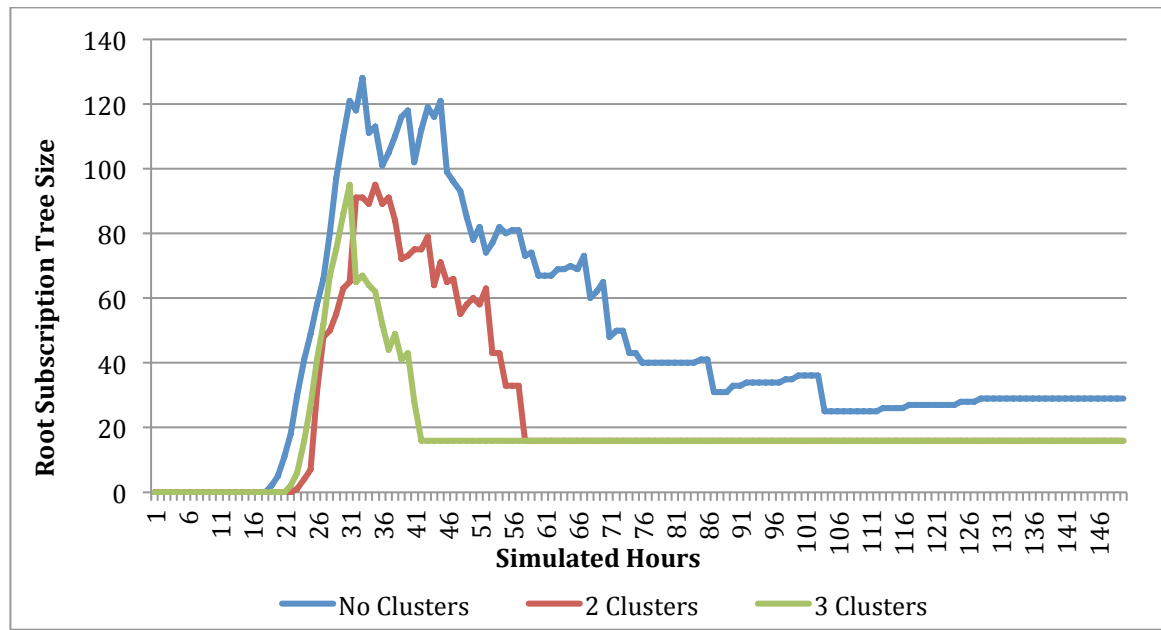


Figure 19: Root Subscription Tree Size on the Master Node (shown in Figure 18)

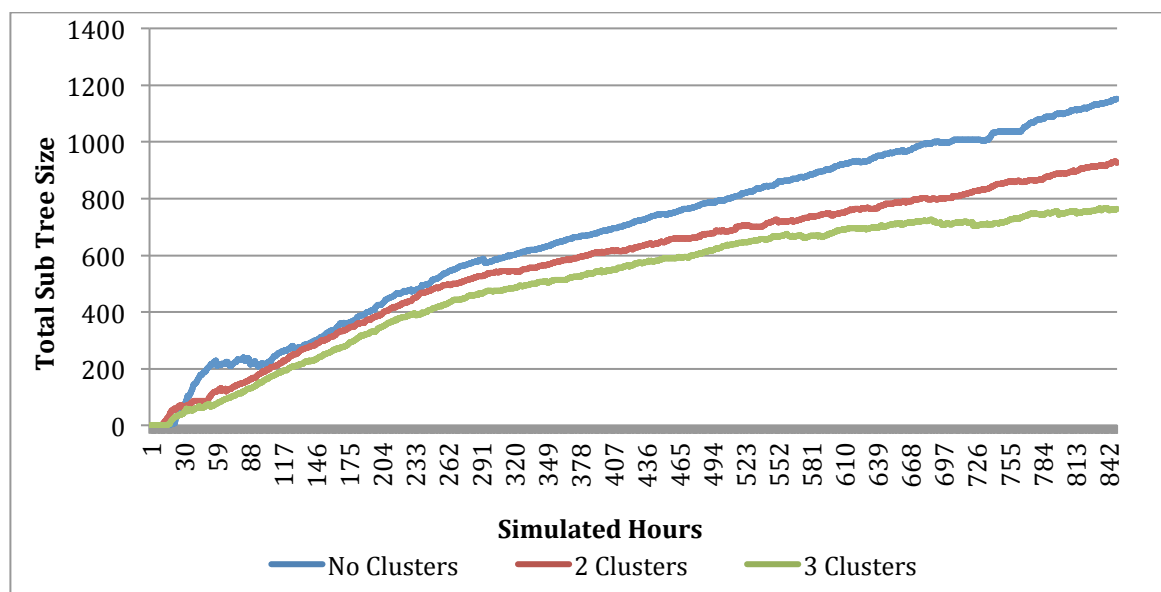


Figure 20: Unique Subscriptions on a Broker on Level 3 (shown in Figure 18)

6.2.5 Conclusion

Figure 19 shows the root subscriptions on the master broker. The growth of the number of root subscriptions is non-linear as this represents the broker merging similar subscriptions as they arrive at the broker into covering relationships over time and hence the fluctuations as subscriptions are received, stored and then merged. This experiment has shown that through the introduction of clustering the subscription table size of brokers decreases.

Shown in Figure 20 are the total subscriptions received by a broker at level three in the broker network. The data shows a gradual rise as subscriptions are received, across the three clustering experiments, although as clusters are introduced the number of unique subscriptions maintained in the broker's subscription table increases sub-linearly. All experiments are presented over a period of simulated hours, subscriptions being clustered over a period of simulation, and different brokers were active during different periods within this simulation, hence the timings are different in each case. Nothing is being varied over time; all that changes is the rate at which various brokers receive subscriptions across the network

This early pilot study into the effects of clustering is presented in [7] and also included in Appendix A. It motivated research into dynamically clustering Knowledge-Based Networks having shown that through the introduction of static clustering the subscription tables become smaller and more optimal as clusters of interest are defined across the network.

However this experiment does not indicate how to handle situations where clients' semantic interests might drift or where the interests of individual clients are not known a-priori, motivating the need for a more dynamic clustering approach, which can be managed in an extensible and flexible manner.

The next section of this chapter provides a detailed evaluation of KBNImpl semantic operator usage costs, subscription tree search times and hop count in delivering messages across clusters before finally evaluating the time taken to travel across a number of hops, where each hop represents a broker in the network.

6.3 KBNImpl Operational Costs

6.3.1 Operator Usage

The first experiment conducted as part of this section looks at the cost of using the various KBNImpl operators in messages, and evaluates the time taken to match incoming publications to stored subscriptions, using the ontology introduced in Section 6.1.1. This experiment was designed to assess how different semantic operators have different time-based costs associated with their use. And the degree to which semantic operators are more costly, in terms of time taken to match incoming publications to stored subscriptions. For this purpose, therefore, only a single broker was needed and used.

Shown in Table 13 is the experimental set-up used in all experiments conducted as part of Section 6.3.1, please refer to Section 6.1.3 for detailed discussion of each of these terms.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:			
1	1			
No of subscribers:	No of subscription filters per subscriber:	No of subscription attributes:	Sub frequency:	
500-3000	2	1	250ms	
No of publishers:	No of publications per publisher:	No of publication attributes:	Pub freq:	No of notifications:
250	2	1	250ms	300

Table 13: Experimental set-up metrics used in experiments

Operator		Subscription	Sample, matching, publication
		<i>Name space, applies to each semantic element = http://confOf#</i>	
Equivalent	@~	Lesson	Tutorial
Not Equivalent	@!~	Lesson	Organization (is not equivalent)
Sub Class	@<	Paper	Contribution
Super Class	@>	Administrative_event	Camera_Ready_event
ISA	@=	City	Dublin
IS NOT A	@!=	Organization	Autonomic_Actions
ONT_PROP	@*	[hasCity, Dublin]	Jones_Dominic
Equal Bag of sub classes @<	#=/@<	[Chair_PC, Administrator, Administrative_event]	[Organization, Person, Event]
Super Bag of sub classes @<	#>/@<	[Chair_PC, Administrative_event]	[Organization, Person, Event]
Sub Bag of sub classes @<	#</@<	[Chair_PC, Administrator, Administrative_event]	[Organization, Person]

Table 14: Example Subscriptions and Matching Publications

Shown in Table 14 are example subscriptions and publications for the data used in this experiment. Each subscription and publication is the same, in terms of the semantic components

above, however each has a unique numerical ID added, to block any covering from occurring, allowing this evaluation to only focus upon the total number of semantic or non-semantic subscriptions held in each broker.

For each publication an additional time-stamp attribute is inserted into the message body when formed. When a notification is delivered to a subscriber, the subscriber is able to use this time-stamp to calculate the time taken to deliver the message, using the time the message was received (in milliseconds) minus the time it was sent.

This new feature of KBNImpl made possible the measurement of delivery time. Operationally, this time measure is only valid when synchronised clocks are guaranteed. For these experiments, this was addressed by running the clients and brokers on the same machine and therefore using the same system clock as shown in Figure 21, the hierarchical topology existing on a single machine for this experiment.

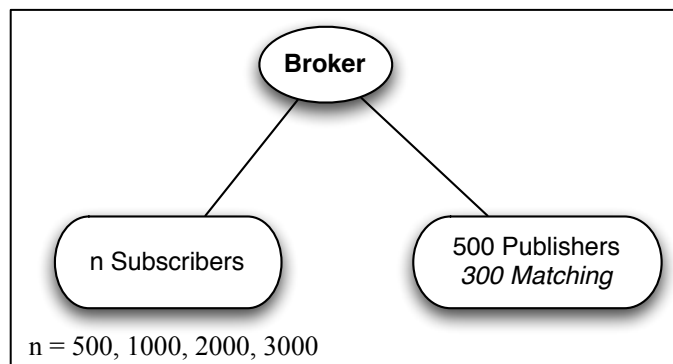


Figure 21: Example Topology, Operator Search Costs

<i>All values in milliseconds</i>	500 Unique Subscriptions		1000 Unique Subscriptions		2000 Unique Subscriptions		3000 Unique Subscriptions	
	Mean:	Std Dev:	Mean:	Std Dev:	Mean:	Std Dev:	Mean:	Std Dev:
Ont Prop	94.76	5.42	185.28	7.15	**	**	**	**
Is A	50.83	12.37	94.64	20.03	190.33	42.35	4319.39	2235.37
Is Not A	50.35	24.76	96.90	40.4	200.83	104.91	4349.53	2078.71
Bag – Super	18.17	6.3	33.45	8.56	62.18	11.86	92.94	16.34
Equiv	17.94	9.67	32.77	14.21	61.76	21.02	90.46	24.8
Bag – Sub	11.40	2.9	20.50	3.95	38.75	4.87	57.60	5.52
Bag – Equal	9.10	3.04	15.98	3.51	30.98	4.77	43.57	4.94
Not Equiv	8.74	4.92	15.37	6.72	27.84	9.78	39.87	11.83
Less Spec	4.47	2.99	7.03	3.46	12.51	3.88	17.11	4.24
More Spec	4.32	2.95	7.61	4.4	11.83	4.14	16.96	4.19

Table 15: KBNImpl Operator Costs – End to end mean delivery times per match (ms)

Shown in Table 15 are the results from this experiment presented from the highest time taken to the lowest. It is clear that the costs associated with using the different ontological operators range greatly in time taken to deliver 300 publications to 300 matching subscribers. The matching time varies sub-linearly with the number of subscriptions. The high values with 3000 subscription present results from the processing time running on after the next publication is received, and

hence two matches being run in parallel. Similarly, the ontological property operator data for the 2000 & 3000 results (shown as **) are excluded, as the cost of searching the ontology for matching subscriptions resulted in Java Out-of-Memory errors on the KBNImpl message broker. This out of memory error was established as a Siena/KBNImpl implementation issue arising from the number of ontological queries that are being passed to the reasoner. This is a KBNImpl bug and not one created through the development of KBNCluster.

This experiment shows the likely variability of performance based on the particular semantic KBNImpl operators employed in the set of subscriptions active at any one time motivating a flexible management approach such that future KBNCluster management policies can be designed to account for the operators in use in a particular sphere of interest.

6.3.2 Subscription Tree Search Time

This second experiment documents the time taken to search a subscription tree/set of various sizes, against incoming publications. This experiment was conducted to establish the relationship between the number of subscriptions on a broker and the time taken to match incoming publications to those stored subscriptions. Additionally this experiment evaluates the effect of searching semantic and non-semantic subscriptions.

Shown in Table 16 are the experimental metrics used in all experiments conducted as part of Section 6.3.2. Again please refer to Section 6.1.3 for detailed discussion behind these terms.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:			
5	1			
No of subscribers:	No of subscription filters per subscriber:	No of subscription attributes:	Sub frequency:	
1000-6000	2	4	250ms	
No of publishers:	No of publications per publisher:	No of publication attributes:	Pub freq:	No of notifications:
250	2	4	250ms	300

Table 16: Experimental set-up metrics used in experiments

Each experiment was run 5 times and from this data the mean value of the overall search time is calculated. Incoming publications were engineered to match 300 subscriptions by the incorporation of a unique ID "*notificationNumber*" in matching pairs. As in the last experiment a time-stamp was added to enable calculation of end-to-end delivery time. Shown in Code Example 14 is an example subscription used in this experiment. Each subscription is made up of a subscription to a class "*Contribution*" and an instance "*Paper.*" In addition to this, each subscription contains a notification number. Using this notification number it is possible to direct which subscriber receives which notification, as shown in Code Example 15, creating static

clusters whilst prompting ontological comparisons to still occur. Matches will only ever occur between the *notificationNumber*, as all ontological filters are kept uniform in this experiment, but will be used in comparison.

Type	Name	Operator	Value
<i>Name space, applies to each semantic element = http://confOf#</i>			
<i>Class</i>	<i>ontClass</i>	@>	<i>Contribution</i>
<i>Instance</i>	<i>ontInstance</i>	@=	<i>Paper</i>
<i>Integer</i>	<i>notificationNumber</i>	=	<i>n...n+1</i>

Code Example 14: Example Semantic Subscription used within this section

Type	Name	Value
<i>Name space, applies to each semantic element = http://confOf#</i>		
<i>Class</i>	<i>ontClass</i>	<i>Paper</i>
<i>Instance</i>	<i>ontInstance</i>	<i>Paper_17</i>
<i>Integer</i>	<i>notificationNumber</i>	<i>n...n+1</i>
<i>Integer</i>	<i>pubTime</i>	<i>N</i>

Code Example 15: Example Semantic Publications used within this section

Figure 22 shows the mean time taken to deliver both non-semantic and semantic messages (where class and instance filters were removed from the subscriptions and publications) across a single broker, where the number of subscriptions varies.

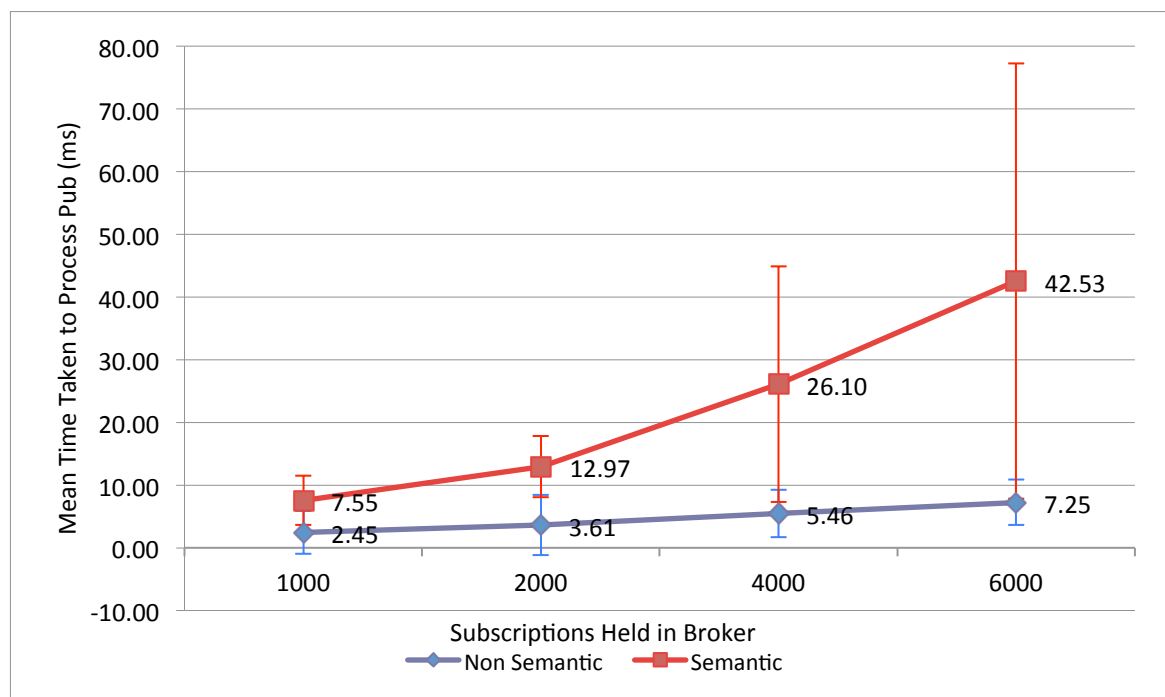


Figure 22: Subscription tree search times (Semantic & Non Semantic)

In conclusion, this experiment has shown that as the number of subscribers increases, so does the mean time to process and deliver the incoming publication to subscribers. It has also shown that semantic subscriptions take more time to search than non-semantic. Also, while semantic subscription matching time increases non-linearly with the increase in subscriptions held, non-

semantic subscription matching time increase linearly. This supports the application of clustering to a KBNImpl by showing that if a subscription tree's size can be reduced or optimised, through clustering and through the introduction of a greater number of root subscriptions, the time to deliver publications to the brokers' attached subscribers can also be reduced.

6.3.3 Hop Count Experiments

This final set of experiments into static clustering looks at the hop count associated with delivering messages, and the time taken to deliver these across a range of hop counts. When defined as a single cluster, publishers and subscribers publish and subscribe to all brokers in the topology. When defined as three clusters, clients are split below the top-level master broker into three clusters and publishers and subscribers manually assigned by the experimenter to specific clusters. A similar process is applied to creating six clusters. To summarise this key difference between the 1, 3 and 6-cluster scenarios is as follows: in the 1-cluster scenario, the consumers and producers are spread randomly across the network, with no correlation between their placement around brokers. In contrast the 3-cluster scenario sees clients statically placed in a specific cluster within the network topology – i.e. nearer to their predicted publishing or subscribing partner, based on directing a client to attach to a broker that may match their interests. The same method is applied in the 6-cluster scenario. Shown in Table 17 are the experimental metrics used in all experiments conducted as part of Section 6.3.3.1 and 6.3.3.2. Again please refer to Section 6.1.3 for detailed discussion behind each of these terms. For both experiments (measurement and timing) please see Code Example 14 and Code Example 15, used in Section 6.3.2, for examples of publications and subscriptions.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:			
<i>1</i>	<i>11</i>			
No of subscribers:	No of subscription filters per subscriber:	No of subscription attributes:	Sub frequency:	
<i>3000</i>	<i>3000</i>	<i>4</i>	<i>250ms</i>	
No of publishers:	No of publications per publisher:	No of publication attributes:	Pub freq:	No of notifications:
<i>3000</i>	<i>4</i>	<i>4</i>	<i>250ms</i>	<i>3000</i>

Table 17: Experimental set-up metrics used in experiments

6.3.3.1 Hop Occurrences

This experiment was conducted to establish whether the number of hops, taken to deliver publications to subscribers could be reduced through clustering conducted to support the arguments behind developing and deploying a dynamic approach to clustering KBNImpl brokers and clients in order to improve efficiency.

In measuring the hop count of messages passing from publisher to subscriber the KBNImpl publication forwarding mechanism was adapted to include a hop count value within the message header itself that is incremented as the message passes across a broker as it traverses the network. In this experiment clustered vs. un-clustered hop counts are measured and publishers and subscribers

are assigned to brokers, based on shared interests, statically set out in code i.e. when they are created as applications they are instructed to connect to a specific cluster or broker. Such an approach is utilised only initially to evaluate such a static approach to clustering. Shown in Figure 23 is a histogram of hop counts in the 1, 3 and 6 cluster experiments. It is clear that as the number of clusters increases, the number of hop counts required for delivery decreases. In a 6-cluster experiment 95% of messages are delivered across 1, 2 or 3 hops, whereas in a 3-cluster experiment that percentage is 80% and in 1-cluster experiment that percentage is only 28%.

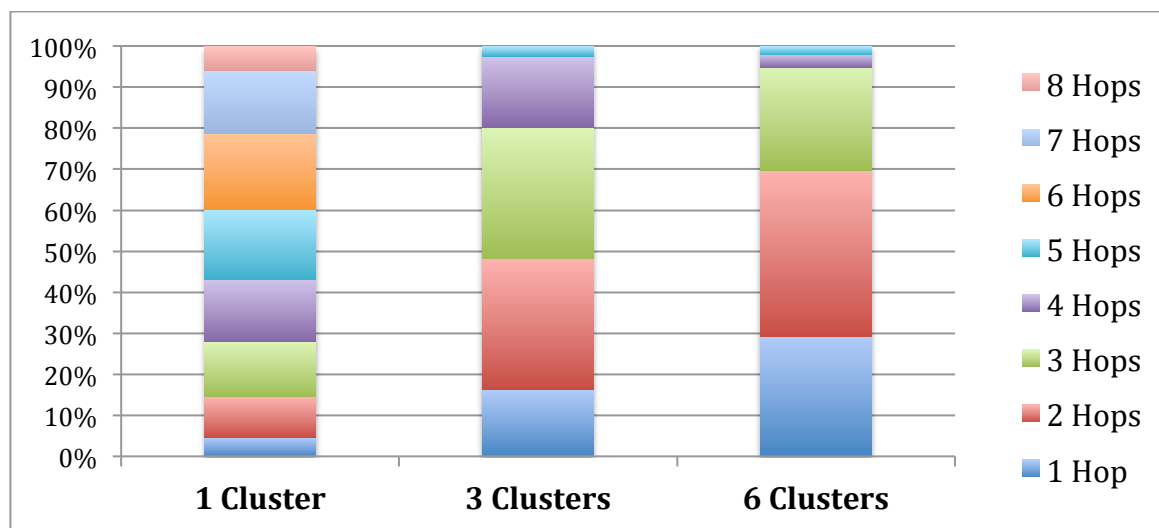


Figure 23: Spread of Hop Counts in Delivered Notifications, across various cluster topologies

In conclusion, this experiment confirms that the number of hops required to route a message decreases as the number of clusters increases. The implication is that a reduction in the number of hops required to route messages from source to destination will have two main impacts: first the broker, upon receiving a publication, has a higher chance of holding a matching subscription to the incoming publication and that, secondly, at a network-level this will decrease the amount of time taken to route messages from source to destination. Fewer hops require fewer brokers involved in the routing of a message and thus a lower load is placed on the whole network's processing capabilities. Supporting the argument behind deploying clustering in KBNCluster as a way to offset some of the costs associated with KBNImpl's increased expressiveness.

6.3.3.2 Hop Count Timing

This experiment was conducted to establish the cost (in terms of time taken) to travel across a varying number of hops in a KBNImpl deployment. Using the data set from the previous experiment, it was possible to calculate the mean delivery time per hop across the network. The notifications received in the previous experiment contain two measureable values, the number of hops they took to be delivered and the time taken to route across those hops. The times per hop, presented in this section, are calculated by grouping all the timings for each hop count and calculating the mean from these values. Shown in Figure 24 is the timing data broken down by publication delivery hop count in each of the three clustering experiments, for publication matching semantic filters. Figure 25 shows the same for **non-semantic** matching filters.

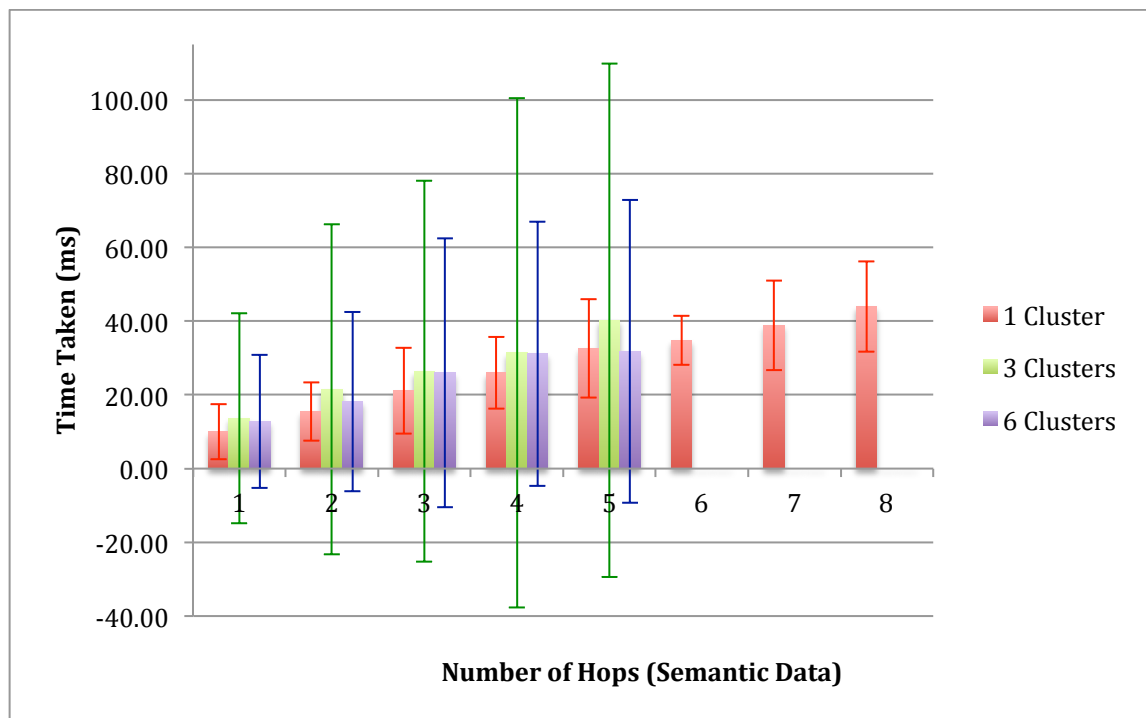


Figure 24: Average Message Delivery Timing, Semantic

In Figure 24 large standard deviations are presented associated with the mean average times taken, per hop for the delivery on semantic messages. In Figure 25 slightly smaller, but still significant standard deviations are presented for non-semantic data. The key reason behind such large variance, in both cases, is due to the number of attributes randomly included in each subscription. Subscriptions with more attributes take longer to process against incoming publications, conversely subscriptions with fewer attributes take less time to process. Each subscription in this experiment was randomly created with between 1 and 4 attributes each, hence timings to delivery each message result in larger variance than expected. The difference between the semantic variance Figure 24 and non-semantic variance Figure 25 is attributed purely to longer time taken to process semantic vs. non-semantic messages. The variance / deviation from the mean, in both cases, is inline with the increased times taken in comparing semantic to non-semantic delivery.

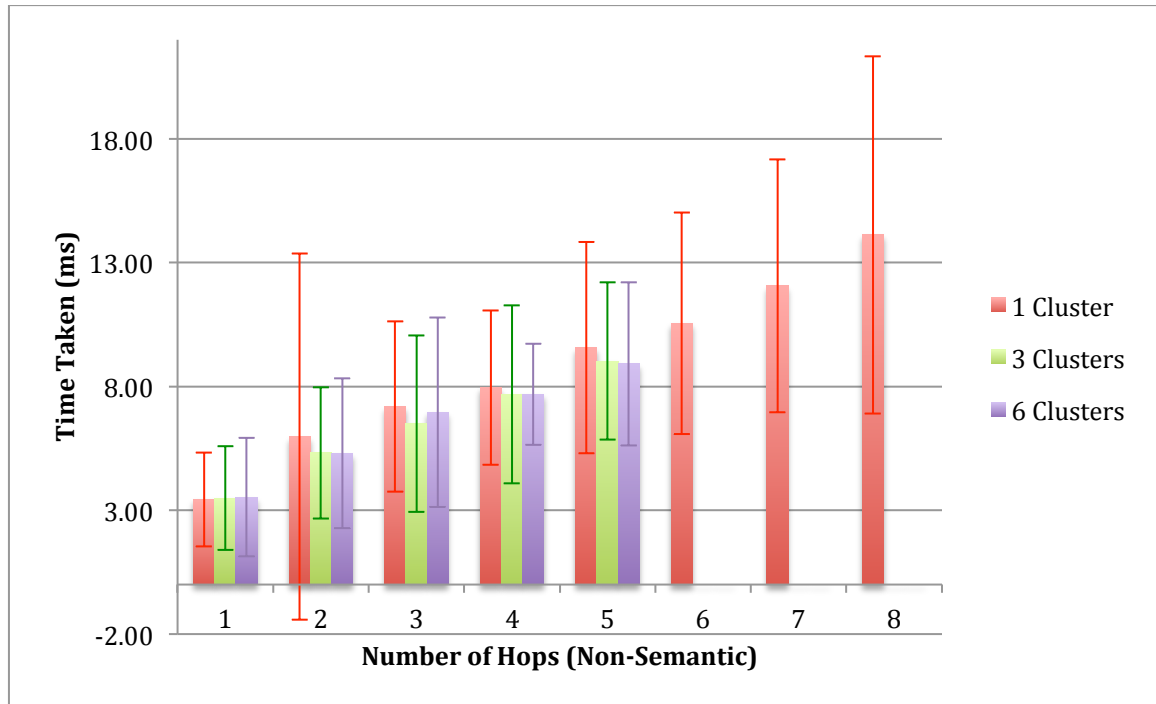


Figure 25: Average Message Delivery Times, Non Semantic

In conclusion this experiment has shown that as the number of hops increases so does the time taken to route over those hops, additionally this experiment has shown that there is an increase in the time taken to process semantic vs. non-semantic. Timing has been shown to be linear with the number of hops; a key premise of this thesis, less hops results in quicker delivery of content from producer to consumer.

6.3.4 Conclusion

This section has established the benefits of **static** clustering KBNImpl publishers, subscribers and brokers. From the evaluation presented a number of conclusions are drawn:

- The introduction of clusters reduces the number of hops between source and destination and correspondingly the time taken to deliver messages.
- Use of semantic publications and subscriptions increases the cost involved in handling messages, esp depending on the operators used. This is a strong argument for the introduction of dynamic managed clustering.

It has been shown in this section that static clustering can be of direct benefit to the publishers, subscribers and brokers of a network through a reduction in load and time taken to route messages. In addition to this it can be inferred that when scaled to multiple clusters, it rapidly becomes unmanageable, due to the number of changes which need to be made, across multiple clients and brokers when operational characteristics change, and the inherent variance in performance between different operators (both semantic and non-semantic).

6.4 Costs Associated with Dynamic Clustering

Static clustering was used in the previous section to evaluate the effects of clustering when applied using a static, manually configured approach which clearly will not scale. Dynamic managed clustering places the logic for guiding clustering outside of the network, thereby removing the requirement for any application specific logic to be placed in the client or broker code. In such an architecture, clients are directed where to attach to the network, based on their semantic interests. As managerial goals or the balance of ontological operators representing those interests change, so the clustering policies can change, without any modification required to the client's or broker's code-base. This section introduces three strands of evaluation, each of which addresses some aspects of the approach taken in the dynamic managed clustering of clients around brokers of common interest, based on the Design and Implementation chapters.

The first evaluation, 6.4.1 (Management Data Storage & Policy Execution Costs) evaluates the costs associated with creating and storing managed objects (MO) in the Management Information Base (MIB) stored on the policy server. This places potential upper limits on the number of brokers, publishers and subscribers that can be managed by a single prototype policy server implementation. In addition to this, the cost of executing policies against those MOs is fully evaluated and presented with regard to time taken, which in turn may constrain the reactivity of the management system to changes. A range of MIBS in the range of 500-2500 are used in each experiment with a 5 policies used in 6.4.1.2, 2 subscriber, 2 broker and 1 publisher.

The second evaluation, 6.4.2 (Data Collection Costs) looks at the costs involved in collecting detailed clustering information from across the broker topology, taking a broker first approach to clustering. This two-phase approach involves the broker calculating its semantic centre (Medoid) or multiple values and returning this value to the policy server. The effect that these management requests have upon normal publish/subscribe operation is important in determining the overall effect that collecting management data (using a broker first approach) has upon the operation of the network.

Finally in the third evaluation, 6.4.3 (Mobility Costs), the various costs associated with moving clients and brokers across the network are evaluated as the policy server makes operational decisions as to where these clients should reside. This section presents the various times associated with each movement. Message delivery is not guaranteed during this period of movement (from start to finish), therefore imposing an operational reliability cost of this management approach. KBNCluster and the evaluation of this thesis works on the premise that from the point at which a client instigates a movement request to the point at which this request is completed messages may not be delivered. This window is used to evaluate the worst-case time taken in moving a client where messages may not be delivered.

6.4.1 Management Data Storage & Policy Execution Costs

The first experiment in this section looks at the cost of creating/updating MO entries in the MIB when incoming management messages are received. There are six different management messages that the prototype policy server (Design Chapter 4, Section 4.4.2) can receive. Three relate to broker state, two relate to subscriber state and one relates to publisher state. They range in the number of attributes they contain, with the smallest containing three attributes and the largest containing ten attributes.

In KBNCluster a single instance of the policy server connects to the trigger broker, as discussed in the Design Chapter 4, Section 4.2.1. For this experiment, the local management agents for simulated brokers, publishers and subscribers connect to this management network and trigger broker, and publish management information towards the policy server, using the trigger broker's semantic delivery mechanism. These messages contain the correct number of attributes, with the correct number of types; however, the actual data used in this experimentation has been simulated, in the range of data values that would be received in real operation. A key design decision of the policy server was that the first message received, which holds a unique and previously unknown client or broker address, represents the first MO of the MIB for each agent; this is updated with new variables as received. This address is either the broker's or a subscriber's KBNImpl assigned address or publisher's uniquely generated ID (UUID). The second experiment in this section looks at the cost of executing, against a variable number of MOs, n policies, where n is fixed but the number of MOs change. This aims to evaluate the efficiency of the policy server against a range of MO in terms of policy execution times and policy server memory usage.

6.4.1.1 Memory Footprint

This first experiment establishes the memory (RAM) required for storing a range of MOs in the MIB. This stress tests the amount of memory required to store and maintain the MIB with an increasing number of MO entries, thereby determining any operational limits on policy server deployments and configuration.

```
System.gc();// Run multiple times
Long startMemory = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
//Do Task - Update / Insert MO data
System.gc();// Run multiple times
long usedMemory = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
long totalMemory = usedMemory - startMemory;
```

Code Example 16: Calculating Memory Usage

Being able to calculate the memory, as shown in Code Example 16, used by the application server over a period of time allows the memory footprint of the application to be determined precisely. The memory usage calculation shown in Code Example 16 is the same as that used by Guo, in KBNMap [4], that being the standard Java library for memory calculation.

In five experiments a range of between 500 to 2500 MOs were sent and initialised by the policy server, with a third of each representing broker, subscriber, and publisher MOs. The standard deviation associated with this data is zero, i.e. regardless of the number of runs which were tested, the data points, which were only measured once did not fluctuate.

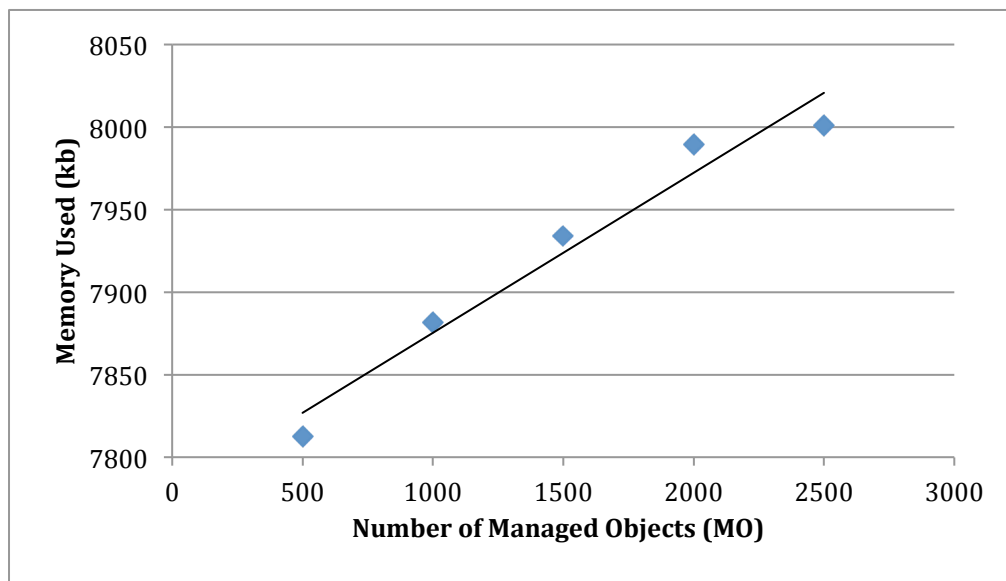


Figure 26: Memory Usage vs. MO Created

In conclusion this experiment has shown, in Figure 26, that the amount of RAM used for storing MOs in the MIB scales well within the experimental bounds of KBNCluster, outlined in Section 6.1.3. This experiment supports the argument that the use of a MIB/MO approach to storing management data, across KBNCluster, is suitable as a buffer between the nodes of the managed overlay and the policy server.

6.4.1.2 Policy Execution Timing

This experiment evaluates the cost of firing policies and having these policies execute against the MIB/MOs previously stored by the policy server. The results from this experiment will establish which policy rule is the most or least costly to invoke, as well as how each rule invocation scales, as the number of MO updates increases.

The experiment is conducted using five policies that fire if, and only if, an update is received (with the correct data set) from a node within the managed overlay. There are three timer-based policies and each operates and fires periodically. They are not evaluated, as they do not exercise the full cycle of events that can trigger a policy evaluation.

The policy rules that are evaluated operate on an *Event-Condition-Action* basis as defined in [19]. In these experiments the *Action* part of the policy is not evaluated. The evaluation focuses on the cost of executing the *Condition* part of the sequence; having received an *Event*. The actions are mostly operated by the management agents on the KBNImpl nodes and are therefore evaluated separately in the remainder of this chapter.

In this experiment the policy server is populated with an equal number of broker, publisher and subscriber MOs. Once populated the full set of subscriber, publisher and broker policies are executed against the MOs. During the experiment the time taken for invoking the policy is measured from the completion of the MO updates to when the last MO has the policy rule condition tested against it. Three types of policies were evaluated in this experiment including:

- Subscriber Policies (See Design Chapter 4, Section 4.5.2):
 - Where should subscribers be placed when they first appear in the network?
 - Does a delivery report identify a subscriber in the wrong cluster?
- Publisher Policies (See Design Chapter 4, Section 4.5.2):
 - Where should a Publisher be clustered when it report its Medoid?
- Broker Policies (See Design Chapter 4, Section 4.5.2 and used to evaluate client- vs. broker-based approach to clustering in the Evaluation Chapter 6 Section 6.4.2):
 - Which is the Broker on which clustering should occur?
 - Which is the Medoid on which clustering should occur?

Detailed explanations of the policies presented above are included in the Design Chapter 4, Section 4.5, and Appendix C of this thesis. Each one of these policy groups is executed against a varying number of MOs, where the number of entries ranges from 500-2500, made up of an equal number of brokers, publishers and subscribers. In line with the last experiment 500 – 2500 MO's were stored in the MIB.

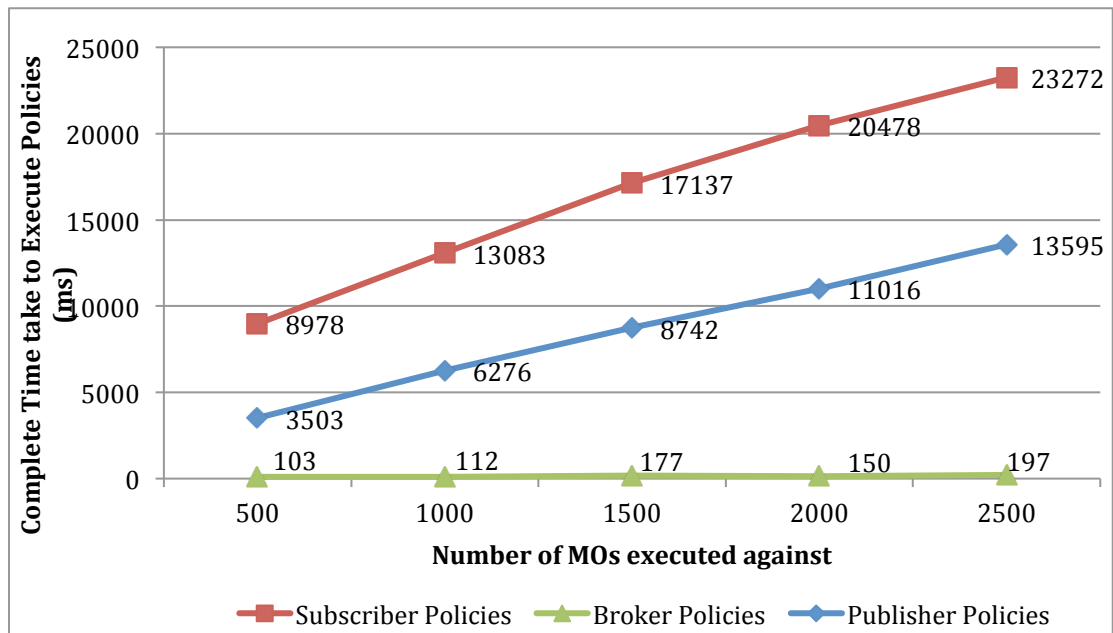


Figure 27: Time Taken to execute a number of MIBS against various Policies

Shown in Figure 27 are the times taken to execute five policies (2 Subscriber, 2 Broker, 1 Publisher rule) against an increasing number of MOs in this experiment. The policy server is bootstrapped with a number of MOs and varied for each experiment. Once populated, the set of policies are executed against the MOs stored and the time taken from the first execution of the first policy to the last is measured. So with 500 MOs populated in the MIB consisting of an equal number of subscriber, broker and publisher MO's, the set of rules are executed against the MIB and the time from start to finish recorded. In stress testing the policy server an equal split between broker, publisher and subscriber was deemed to represent the best split between MO entries in the MIB. The standard deviation associated with this data is zero, i.e. regardless of the number of runs which were tested, the data points, which were only measured once did not fluctuate.

Figure 27 shows a linear growth in each case for the time taken to process policies against an increasing number of MOs. Looking at the broker policies, we see a very small increase in time taken as the number of MOs increases. Broker policies take an array of MOs representing the brokers as an input variable, and the Drools [49] rule engine searches this array for any individual MOs that match the conditions outlined in the policy conditional statement.

In conclusion this experiment has shown that as the number of MO entries increases so does the time taken to execute the set of policies against the MIB. Also shown are the different execution times of various policies, these differences having been previously discussed. This experiment, much like the experiment shown in Section 6.4.1.1 confirms the feasibility of the management approach applied in KBNCluster. The upper value for the policy execution time is appreciable, and therefore will need to be taken into consideration when configuring the period with which MO data updates are requested from the KBNImpl nodes. Feasibility is assured through providing upper bounds of the rates of MO's that can be stored by the policy server. These are well within

the experimental bounds of KBNCluster. Further engineering of the policy server could increase this value, the number of MO's possible to store, as needed. This experiment has shown that the implemented policy server can perform in line with the experimental bounds of this thesis. It does not evaluate the policy server at an Internet scale.

6.4.1.3 Conclusion

In this sub-section the first experiment 6.4.1.1 (Memory Footprint) has shown that the memory usage of the policy system is only slightly affected by the cost of processing and storing a wide range of incoming MO messages. The memory footprint of the MIB rises only 0.2MB when the number of incoming messages is increased from 500 to 2500 (Figure 26). It is concluded from this data that the cost of storing MOs on the policy server only increases slightly as the number of MOs increases and scales well.

From the second experiment 6.4.1.2 (Policy Execution Timing) it is concluded that using the Drools rules accumulate and iterate functions in rule operation greatly out-performs passing single objects to the rule engine.

However, regardless of the way in which the objects are processed, the processing time scales linearly with the number of rules and number of MOs being evaluated, though with non-trivial execution delays emerging, which must be considered in the operational configuration of KBNCluster.

6.4.2 Data Collection Costs

KBNCluster provides thirteen sample policies for clustering, included in Appendix C. Two of these thirteen policy rules require management information requested (by the policy server) from the brokers themselves and are stored as MOs on the policy server. This section evaluates the cost of collecting this management information from these brokers.

The policy server subscribes to the trigger broker with subscriptions to future management updates, from the brokers, publishers and subscribers. There are two management request messages that could be utilised in clustering and that require the broker to publish management information regarding their operational status. These are defined as Management Message 1 (MgntMsg1) and Management Message 2 (MgntMsg2). In this section, the cost of requesting these messages is evaluated and both messages are introduced in the next two sub-sections in turn. Three values are evaluated across a broker network, while a range of management messages are requested, in order to assess the impact on regular KBNImpl operation while supporting the management data monitoring needed for KBNCluster policy management. These are:

- **Average subscription processing times:** The time taken to process incoming subscriptions and store these in a broker's subscription tree.
- **Average publication processing times:** The time taken to process incoming publications against a number of stored subscriptions.
- **Average end-to-end delivery:** The time taken from a publication's submission through to its delivery to an interested subscriber.

The next two sections (6.4.2.1 and 6.4.2.2) introduce Management Message 1 and Management Message 2 respectively. Following this sections (6.4.2.4, 6.4.2.5 and 6.4.2.6,) evaluate subscription processing times, publication processing times and end-to-end delivery times respectively against the two types of management message. Section 6.4.2.3 dealing with Standard Error calculation, based on the set of standard deviations.

6.4.2.1 Management Method 1

Action required by the broker:

```

Create MedoidCollectionAL ArrayList:
Create SET of ALL Subscribers (where semantic subscribers)
For (each in SET){
    Create new SemanticSubObj (Subscriber)
    Create new RepresentativeMedoid (SemanticSubObj);
    Add RepresentativeMedoid to MedoidCollectionAL ();
}

```

Code Example 17: Request Medoid

Shown in Code Example 17 is the pseudo-code representation of what is executed when a request for MgntMsg1 is received by a broker. This requires a broker to enumerate their complete set of semantic subscribers into a number (n) of distinct representative Medoids where n is dictated by the policy server. This collection of representative Medoids is calculated by grouping and pair-wise merging the semantic representation (Medoids) of the brokers semantic subscribers, as shown in Code Example 18.

```

numbOfRepresentativeMedoids definedBy POLICY_SERVER;
MedoidCollectionAL defined in Previous Step;
While (MedoidCollectionAL > numbOfRepresentativeMedoids){
    Start = Rep Medoid with smallest average semantic Spread;
    //Remove Start from MedoidCollectionAL;
    End = Rep Medoid with closest Medoid to Start;
    //Merge the two representative Medoids.
    Start.merge (end);
}

```

```

        Remove (end) ;
        MedoidCollectionAL.insert (Start) ;
    }
    return MedoidCollectionAL;

```

Code Example 18: Reduce down to N clusters

Each response to the policy server, by each broker, contains the cluster's name, the cluster's Medoid and associated semantic spread of this Medoid. This information represents the broker and all of the broker's clients in a single Medoid. The semantic spread of the Medoid is the standard deviation of the mean average from the Medoid to all other entries in a set of query values. With this information the policy system is able to identify the semantic Medoid or semantic make-up of every broker in the network, used if a broker-first approach to clustering is utilised.

6.4.2.2 Management Method 2

Action required by the broker:

```

    Using the process outlined in POLICY_2 calculate 1 Cluster;
    Also Calculate brokers:
        Hierarchical Server location;
        Number of Active Subscriptions;
        Number of Notifications Delivered;
        Single Clusters Medoid;
        Single Clusters Medoid Spread;
    Return all values to Policy Server;

```

Code Example 19: Request Medoid Info

The second message request to the broker is defined as Management Message 2 (MgntMsg2) shown in Code Example 19. MgntMsg2 requests utilisation reports from brokers, and contains information about the brokers, subscribers, publishers and notification delivery metrics of the broker as well as a single Medoid for the broker and its associated standard deviation. This is, in broker first clustering, used to calculate the accuracy of the client's placement with that broker.

In the rest of this section the costs involved in calculating the average publication processing time, average subscription processing time, and end-to-end delivery time, is measured, whilst a range of management message 1 and 2 are requested by the policy server from a broker over a period of experimentation.

6.4.2.3 Standard Error Calculation

In subsequent experiments (6.4.2.4, 6.4.2.5 and 6.4.2.6) the mean of mean calculations from five experimental runs are shown in Figure 28, Figure 29 and Figure 30. The standard error for each of these data points is calculated, as below, against the set of standard deviations.

$$\text{SUM (SET OF STANDARD DEVIATIONS) / Square Root (Num data points)}$$

The above calculation provides the standard error variation from the mean of means for each of the experiments and is seen as being the sampling variance of the complete set of standard deviations. The formula used in calculating the standard error can be referenced in [53].

Where a single standard deviation represents the variance in a mean value of a set of data points, the standard error shows both the variance across the set of means and standard deviations. From this standard error calculation, it is possible to assess the accuracy of the mean of means, based on the difference between the mean and the standard error (both plus and minus).

This calculation, and the standard errors for each experiment are shown in Table 19, Table 21 and Table 23 for each of the experiments conducted as part of this section.

6.4.2.4 Subscription Processing Times

This experiment looks at the overhead involved in processing and storing subscriptions whilst the policy server is requesting a range of management messages. This experiment was designed to determine how the time taken to process and store incoming subscriptions in the KBNImpl broker scaled as the rate of management requests increased. Shown in Table 18 are the experimental metrics used in all experiments conducted as part of this section. Again please refer to Section 6.1.3 for detailed discussion behind each of these terms.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:			
5	1			
No of subscribers:	No of subscription Filters:	No of subscription attributes:	Sub frequency:	
500	1	2-5	9ms	
No of publishers:	No of publications:	No of publication attributes:	Pub freq:	No of notifications:
N/A	N/A	N/A	N/A	N/A

Table 18: Experimental set-up metrics used in experiments

The policy server requests n management messages (alternating between the 2 types of management messages) from the broker over a period of one minute, as publications are also sent into the broker. The broker receives no incoming publications during this experiment; the only measurement is the storing of incoming subscriptions.

Shown in Figure 28 and Table 19 is that the mean time taken to process incoming subscriptions increasing from around 6.6ms when no management messages are being requested to around 20ms when 10 management messages per min are being requested. Showing that as more management messages are requested the speed at which incoming subscriptions are processed decreases.

It is clear that, as management requests increase, the subscription processing performance of the KBNImpl broker decreases, indicating the performance overhead that this management function imposes on the core network well above any acceptable threshold.

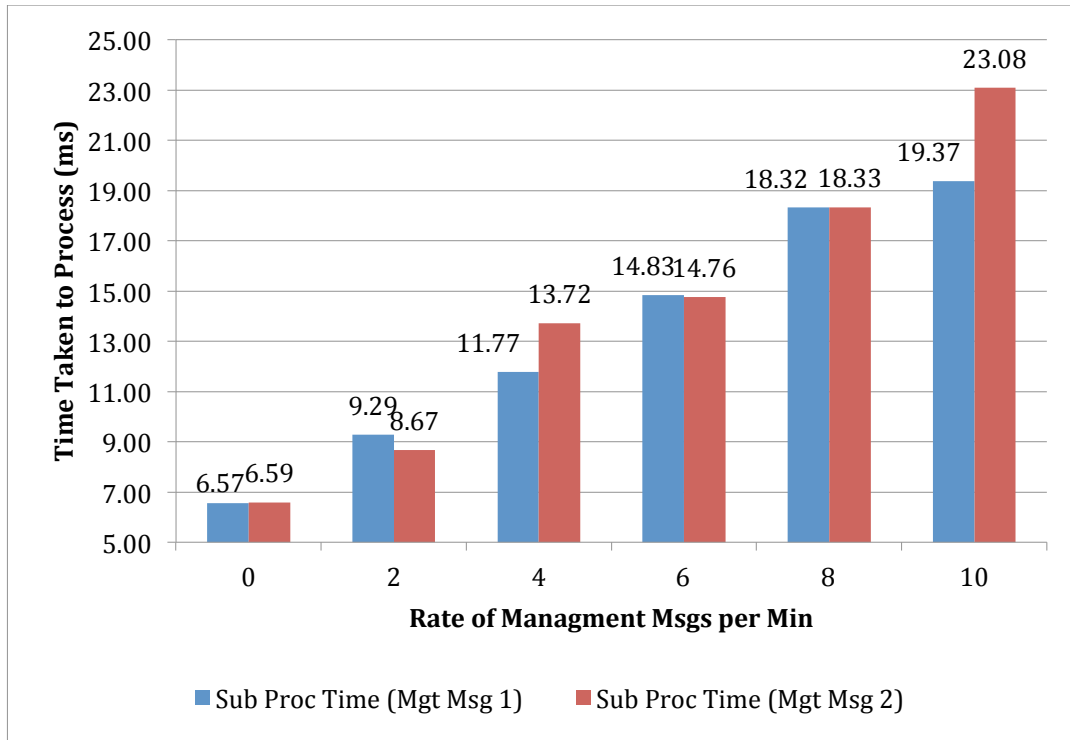


Figure 28: Average Subscription Processing Times (ms)

Number of Mgmt Msg Requested per min					
0	2	4	6	8	10
Management Message Type 1 - Standard Error					
1.65	11.33	15.91	22.94	29.40	26.57
Management Message Type 2 - Standard Error					
3.98	55.46	44.83	63.45	70.57	81.64

Table 19: Standard Error Subscription Processing Times (ms)

6.4.2.5 Publication Processing Times

This experiment was conducted to establish the effect that requesting management information has upon the time taken by a broker to process publications against stored subscriptions, in KBNImpl. The processing time measured is calculated from the point when a publication is received to when it has been matched against the complete set of subscriptions held by a broker. This experiment was designed to determine, in line with the previous experiment, the impact on the publication processing time taken by a broker as management message requests increase. Shown in Table 20 are the experimental metrics used in all experiments conducted as part of this section.

In the bootstrap phase of this experiment a single broker is populated with a number of subscribers. The policy server and publisher client both then start at the same time. The policy server requests n management messages (alternating between the 2 types of management messages) from the broker over a period of one minute, as publications are sent into the broker.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:			
5	1			
No of subscribers:	No of subscription Filters:	No of subscription attributes:	Sub frequency:	
500	1	2-5	9ms	
No of publishers:	No of publications:	No of publication attributes:	Pub freq:	No of notifications:
120	1	2-5	250ms	35

Table 20: Experimental set-up metrics used in experiments

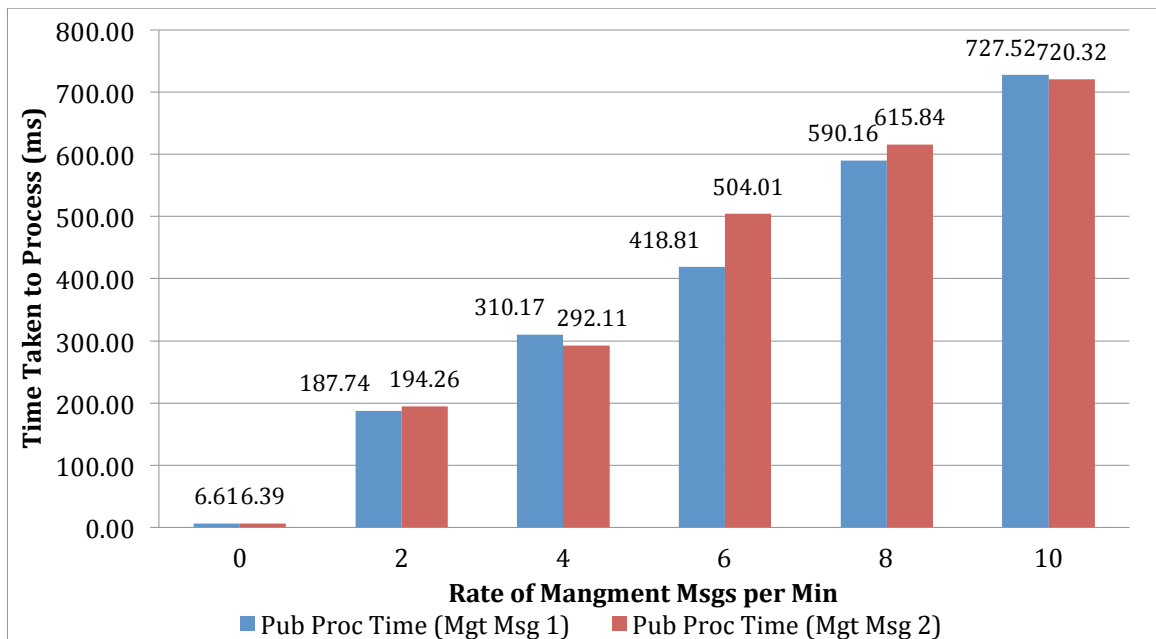


Figure 29: Average Publication Processing Times (ms)

Number of Mgmt Msg Requested per min					
0	2	4	6	8	10
Management Message Type 1 - Standard Error					
3.28	253.15	313.50	347.56	388.20	409.66
Management Message Type 2 - Standard Error					
3.03	258.92	299.95	388.98	401.77	408.73

Table 21: Standard Error Publication Processing Times (ms)

Shown in Figure 29 and Table 21 is the baseline measurement (with no management messages being requested) of around 6 milliseconds to process a publication, which grows to around 700 milliseconds when 10 management messages are requested over a period of one minute. This is roughly a 100-fold increase and shows the load placed upon the broker publication processing functions while calculating and delivering management information. In line with the previous experiment this set of data shows a large increase in the time taken to process incoming publications against stored subscriptions, as they arrive at the broker, whilst two different management requests are being made from the broker. Each time a management message is requested the broker has to perform multiple measurements of their subscription tree, in

calculating the metrics required to fulfil the request for management information. These requests have a clearly huge and negative impact upon the performance of the broker. A 100-fold increase in publication processing times when 10 management requests are requested is unjustified in any performance analysis. This is especially true when compared with the results of Section 6.4.1.3 which shows that management request processing has a much larger impact on publication processing time than sub processing as the broker is both processing incoming publications against stored subscriptions and calculating the response to management message requests, well above any acceptable threshold. However in a final experiment conducted as part of this subsection publication to subscription delivery times are measured.

6.4.2.6 Pub-to-Sub Delivery Times

The previous experiment looked at the time to process incoming publications against stored subscriptions in the broker itself. This experiment evaluates the end-to-end delivery time of a number of publications through a broker while a varying number of management messages are being requested. The period is measured from the point at which the publication was created in the publishing client to it being delivered to the subscriber. This experiment, in line with the two previous experiments is designed to determine the impact on the total time taken to deliver publications from publishers to subscribers, as the rate of management message requests also increases.

Shown in Table 22 are the experimental metrics used in all experiments conducted as part of Section 6.4.2.6. In the bootstrap phase a broker is populated with n subscribers. Once populated, publications are submitted over a period of one minute whilst n' management messages (alternating between the 2 types of management messages) are requested from the broker over the same period, as publications are also sent into the broker.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:			
5	1			
No of subscribers:	No of subscription Filters:	No of subscription attributes:	Sub frequency:	
500	1	2-5	9ms	
No of publishers:	No of publications:	No of publication attributes:	Pub freq:	No of notifications:
120	1	2-5	250ms	35

Table 22: Experimental set-up metrics used in experiments

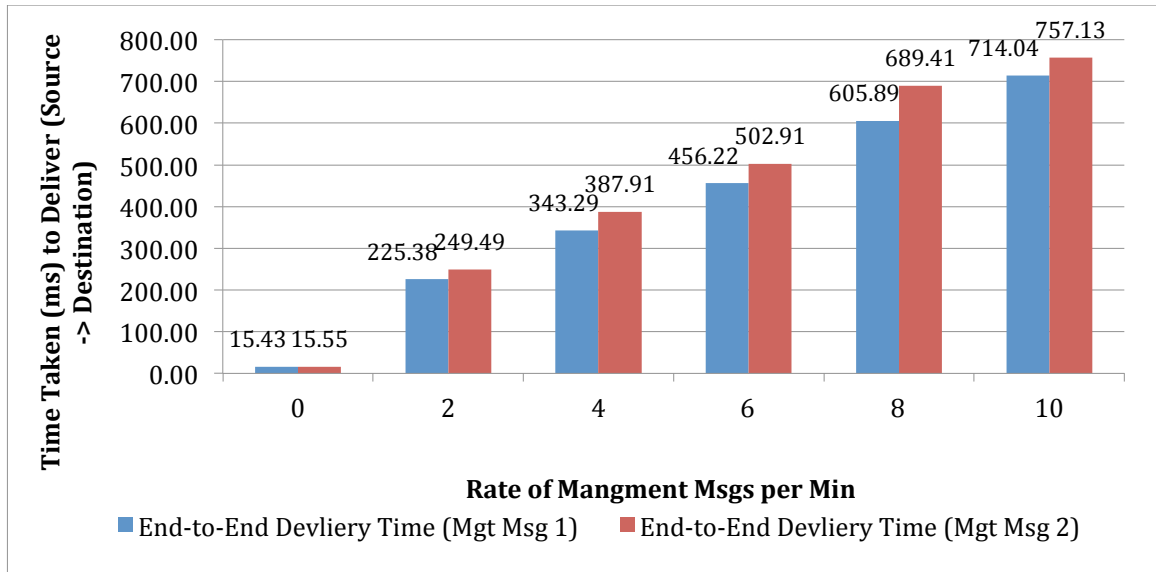


Figure 30: End-to-End Delivery Time (ms)

Number of Mgmt Msg Requested per min					
0	2	4	6	8	10
Management Message Type 1 - Standard Error					
7.04	258.89	314.09	358.83	392.55	399.27
Management Message Type 2 - Standard Error					
8.63	271.38	343.20	358.98	396.98	406.80

Table 23: Standard Error End-to-End Delivery Times (ms)

Figure 30 and Table 23 show an increase in end-to-end delivery costs, consistent with the trend observed for average publication processing times shown in Figure 29. Again this experiment has shown that as the rate of management messages increases, so does the time taken to deliver messages from publisher to subscriber, across the broker on which management requests are being made. Therefore management message requests are shown to degrade performance outside of any definition of acceptable performance.

6.4.2.7 Conclusion

This section of evaluation has shown that ten management messages requested from the broker network over the period of one minute results in:

1. An increase in the cost of the subscription processing time (of around 20ms).
2. An increase in publication processing time (of around 700ms).
3. An increase in end-to-end publication to subscription delivery time (of around 700ms).

From this it is concluded that the costs involved in subscription and publication processing and end-to-end delivery increase unsatisfactorily. Even with 2 management requests per minute data performance impacted severely upon performance. From evaluation it was shown that there was no justifiable rate at which management message requests could be satisfied. The data requests were unacceptable at any rate of management requests. Another approach was required.

Such increases in costs led to this work and the KBNCluster implementation taking a client first approach to clustering, moving subscribers and publishers around static clusters of brokers, dividing and sharing these costs between clients. In KBNCluster Subscribers have their Medoids calculated for them, by message brokers, as they attach to the broker whereas publishers calculate their own Medoid and report this back, to the policy server, for clustering.

The reason for this design decision was the rapidly increasing costs associated with the normal operation of the message broker as a number of management messages requested increased. Rather than one broker calculating the Medoid of multiple clients on request of the policy server, the subscribers are processed as they arrive at the broker, typically a relatively rare event, and all publisher calculations, which occur relatively frequently on the publishers themselves. Thus this management overhead is pushed onto the relatively more abundant client resources, rather than the scarcer broker resources, where the impact on the general performance level of a single Medoid calculation will be greater.

In the next section of this thesis the time costs involved in introducing mobility to brokers, subscribers and publishers are evaluated. This explores how the introduction of movement methods relates to the operational performance of a broker network.

6.4.3 Mobility Costs

KBNImpl operates in a relatively static operational configuration, resulting from the underlying routing mechanism of the Siena CBN on which it is built. This in functional terms makes it very difficult for the subscribers and brokers of the network to be mobile. For dynamic clustering to occur a broker needs to be able to move itself across the network, move its subscribers to other brokers, and merge itself with other brokers across the network. In the KBNImpl code base, there existed a single method by which a broker and all the broker's attached subscribers could be moved. However the following methods have been implemented as part of KBNCluster:

1. Move whole broker and all subscribers to become a child of another broker.
2. Move one, some or all subscribers, using their ID, to another broker, a new KBNImpl feature.

However with both of the above and throughout this complete section the following assumption is made: *When a movement process is instigated, message delivery is not guaranteed from the point the movement process begins until it completes.* This is true as publications travel upwards to the root node of the broker hierarchy and then back down towards child nodes, as necessary. While a subscriber is moving from branch to branch, they may miss publications that have already traversed the branch to which they are moving. This is a worst-case scenario. However, use of the above presumption allows us to measure performance of movement unambiguously with respect to message delivery. Below are example Subscriptions used in this experiment

```
( OntClass0 @<"http://confOf#Poster" OntClass1 @>"http://confOf#Author" ontInstance2
@="http://confOf#Administrative_event" ontInstance3
@="http://confOf#Reviewing_results_event" ontInstance4
@="http://confOf#Submission_event" )

( OntClass5 @<"http://confOf#Short_paper" ontInstance6 @="http://confOf#North_America"
ontInstance7 @="http://confOf#Workshop" )

( OntClass8 @>"http://confOf#Social_event" OntClass9
@<"http://confOf#Administrative_event" OntClass10 @>"http://confOf#Asia" ontInstance11
@="http://confOf#Poster" ontInstance12 @="http://confOf#Student" ontInstance13
@="http://confOf#Registration_of_participants_event" )

( OntClass14 @>"http://confOf#Camera_Ready_event" ontInstance15
@="http://confOf#Member" ontInstance16 @="http://confOf#Member" )

( OntClass17 @>"http://confOf#Europe" ontInstance18 @="http://confOf#Organization"
ontInstance19 @="http://confOf#Reception" )
```

Code Example 20: Example Subscriptions used within the Section

6.4.3.1 Moving Broker & Moving All Subscribers

This experiment was conducted to establish which of the two approaches was the most cost efficient: moving a broker and all attached subscribers or moving all subscribers individually.

Each movement time is measured, as the period from when a broker is instructed to move until the time the action is fully complete. Shown in Table 24 are the experimental metrics used in all experiments conducted as part of Section 6.4.3.1. Code Example 20 presents example subscriptions used across this experiment.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:			
5	2 (<i>Source and Desintation</i>)			
No of subscribers:	No of subscription Filters:	No of subscription attributes:	Sub frequency:	
200-1000	2	1-6	9ms	
No of publishers:	No of publications:	No of publication attributes:	Pub freq:	No of notifications:
N/A	N/A	N/A	N/A	N/A

Table 24: Experimental set-up metrics used in experiments

Shown in Figure 31 are the times taken to move a broker and all attached subscribers compared to the times taken to move just subscribers. When moving the broker and all subscribers, a single instruction is issued to the broker and the broker un-subscribes all top-level subscribers from its parent and moves its master link to its newly defined master. In contrast to this when moving all subscribers the broker instructs each subscriber to move individually and where to move to. The subscribers then instigate this movement by themselves.

The slightly non-linear growth, in both plots, is attributed to the random distribution (1-6) of the attributes contained in the complete set of subscriptions, used in each experiment. In some experiments more subscribers with more subscription attributes could be moved in less time, due to the smaller number and less complex subscriptions of those subscribers, hence non-linear growth.

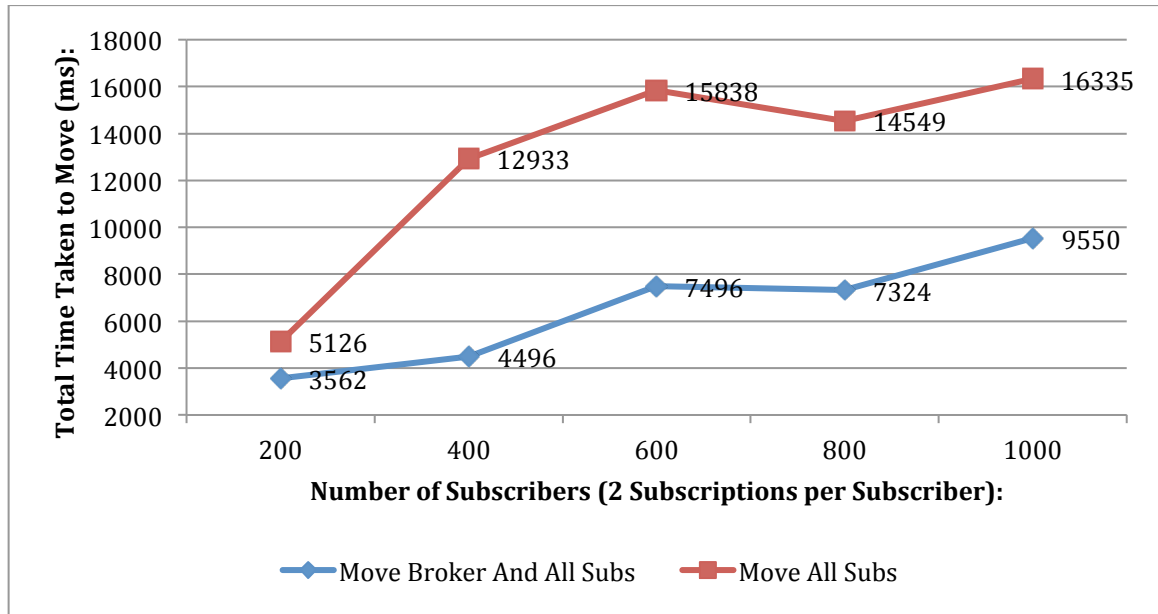


Figure 31: Moving Broker & All Subs and Moving All Subs Individually

Num of Subscribers:	Mean Time to Move Broker + Subs (ms):	Std Dev:	Mean Time to Move All Subs Individually (ms):	Std Dev:
200	3562	387.96	5126	979
400	4496	327.45	12933	1367
600	7496	680.15	15838	1854
800	7324	877.27	14549	1384
1000	9550	1011.75	16335	1286

Table 25: Experimental Data

In conclusion, this experiment has shown, as shown in Figure 31, that it is more costly, in terms of time, to move each subscriber attached to a broker individually than it is to move the whole broker and all subscribers at once. This is attributed to the fact that the broker can move all the subscriptions of its attached subscribers in one go, as root subscriptions, to a new master, keeping its connections to all of its individual subscribers alive as opposed to subscribers un-subscribing, changing master brokers, and re-subscribing, for each subscriber in turn.

This experiment supports a broker-based approach to KBNCluster where the last set of experiments (Section 6.4.2) supports a client-first approach. Even though this is true, in the design of KBNCluster it was decided that the management message collection costs, previously evaluated, were too high to incorporate into implementation in comparison to the difference between a broker-first and client-first approach to movement. Therefore KBNCluster takes a client (subscriber-based) approach to movement, rather than a broker (and all subscribers) based approach, due to the substantial decrease in the performance of the KBNImpl, when management messages are requested during operation. In summary, increases in subscriber- to broker-based movement costs are less than costs involved in collecting the management data used in clustering brokers, as shown in the last experiment (Section 6.4.2) and so incorporated into design.

6.5 Dynamic Clustering Evaluation

Clustering has previously been defined as the process of grouping publishers and subscribers around brokers that share common interests. The static process previously evaluated involves each client being manually configured by a network manager to be placed into the most suitable cluster. The concept of managed dynamic clustering involves the dynamic placement of publishers and subscribers into the most suitable clusters. In KBNCluster this process is performed by the policy server, providing managed dynamic clustering, evaluated in this thesis.

In this section a managed, clustered topology (KBNCluster) is compared to an un-clustered topology (KBNImpl). The network of eleven Brokers used within all experiments, is shown in Figure 32.

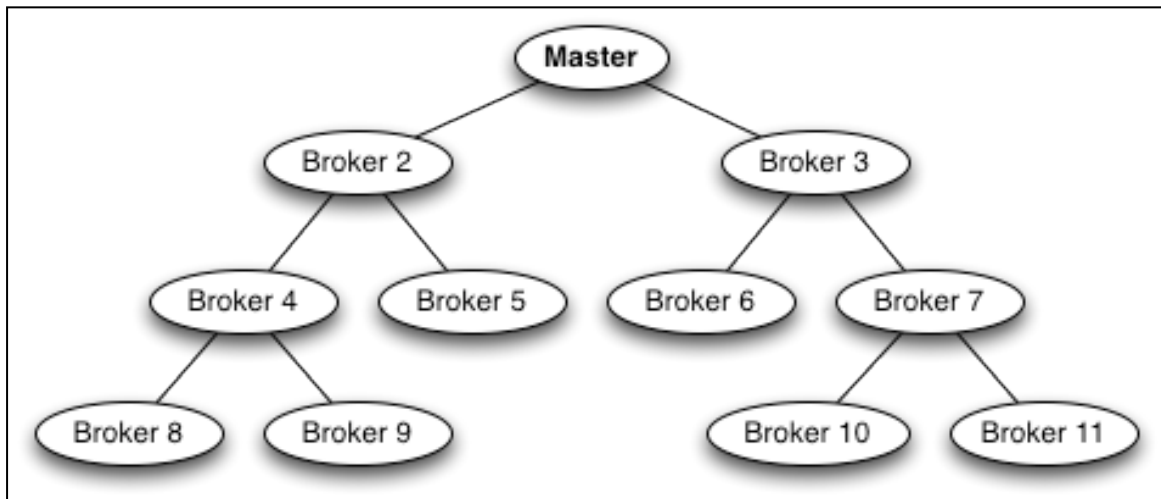


Figure 32: Dynamic Clustering - Server Topology

In both clustered (KBNCluster) and un-clustered (KBNImpl) experiments the same set of publications and subscriptions are used, in the same order of usage, and with the same clients, included in Appendix D. For both the clustered and un-clustered systems the following three key markers of performance are evaluated:

- A smaller more ordered subscription set is easier to search, particularly with more complex ontological subscriptions,
- With a smaller hop count, a message is delivered to its destination by travelling through fewer brokers, thereby reducing overall routing overhead,
- Finally clients residing in the least suitable cluster can be identified, via the number of hops taken to deliver their messages and re-clustered.

6.5.1 Experimental Metrics

Shown in Table 26 are the experimental metrics used in all experiments conducted as part of Section 6.5.2, 6.5.3 and 6.5.4. In the first experiment 6.5.2 no publications are used as only subscription tree size is being evaluated.

Shown in Code Example 21 and Code Example 22 are example subscriptions and publications used throughout this section. Real user data sets of publish/subscribe systems are hard, if not impossible, to come by, hence the evaluation conducted as part of this section does not manipulate or manage semantic similarity between the elements of subscriptions. Subscriptions are randomly created from attributes sourced across the ontology so that any experimental bias in the subscription sets is removed.

EXPERIMENT SET-UP METRICS				
No of runs:	No of brokers:	Number of Clusters = 11.		
1	11			
No of subscribers:	No of subscription Filters:	No of subscription attributes:	Sub frequency:	
250	2	1-6	9ms	
No of publishers:	No of publications:	No of publication attributes:	Pub freq:	No of notifications:
1000	1	1-5	250ms	98

Table 26: Experimental set-up metrics used in experiments

```
( OntClass @<"http://confOf#Reviewing_results_event" OntClass @<"http://confOf#Banquet"
ontInstance @="http://confOf#Reception" ontInstance @="http://confOf#Contribution" )
( OntClass @>"http://confOf#South_America" ontInstance @="http://confOf#Trip"
ontInstance @="http://confOf#Topic" )
( OntClass @<"http://confOf#Person" OntClass @>"http://confOf#Tutorial" ontInstance
@="http://confOf#Africa" ontInstance @="http://confOf#University" )
( OntClass @<"http://confOf#Contribution" ontInstance @="http://confOf#Event"
ontInstance @="http://confOf#Conference" )
( OntClass @>"http://confOf#Author" OntClass @<"http://confOf#Country" ontInstance
@="http://confOf#Asia" ontInstance @="http://confOf#Assistant" ontInstance
@="http://confOf#Reviewing_event" )
```

Code Example 21: Dynamic Clustering, example subscriptions

```
( OntClass ="http://confOf#Administrative_event" ontInstance="http://confOf#Short_paper_19")
( OntClass ="http://confOf#North_America" ontInstance="http://confOf#Fomal_Methods")
( OntClass ="http://confOf#City" ontInstance="http://confOf#Carlow")
( OntClass ="http://confOf#Administrator" ontInstance="http://confOf#Lewis_Dave")
( OntClass ="http://confOf#Contribution" ontInstance="http://confOf#Dynaism")
```

Code Example 22: Dynamic Clustering, example publications

6.5.2 Subscription Tree Size

In this experiment the subscription set of brokers when clustered (KBNCluster) and un-clustered (KBNImpl) are evaluated. In this experiment the *root subscription tree size* is evaluated representing the total number of subscriptions, which cannot be merged into more general, root

subscriptions in both a clustered (KBNCluster) and un-clustered (KBNImpl) topology. This experiment was conducted to establish whether, when clustering is applied to the broker network, the number of root subscriptions received by the network of brokers is reduced. In this experiment subscriptions and publications are sent to specific brokers either via an un-constrained choice (KBNImpl) or via a dynamic managed choice (KBNCluster). The attributes of the messages were randomly chosen, in that there was no relationship between similarity of concepts in publications or subscriptions; they were randomly selected from the source ontology.

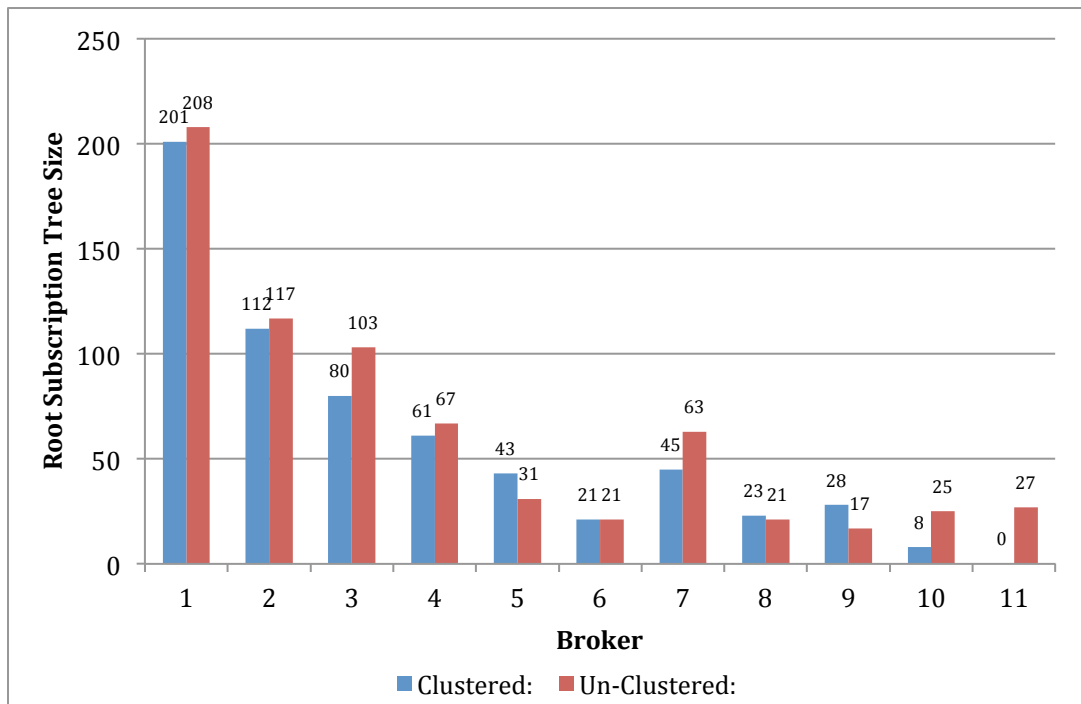


Figure 33: Root Subscription Tree Size (Clustered and un-clustered)

Shown in Figure 33 are the root subscriptions of each of the eleven brokers in the broker hierarchy shown in Figure 32. Root subscriptions are on average 2 messages less per broker, when clustered, than in comparison to total subscriptions. This low level of clustering is established as a worst case result attributed to the random choice of subscription filters, where subscriptions do not cover all or more of the attributes of other subscriptions. The numbers of attributes within each subscription was limited to between 1-5 randomly chosen values. Such **restrictions** and **randomness** in the subscription set results in a much lower occurrence of covering, or chances of covering occurring, in comparison to real user subscribers, which are less random in nature and which share more common concepts.

It is important to note that the level of covering shown in this experiment is smaller than that present in Experiment 6.2 (Static Approach to Clustering) where in motivating clustering more emphasis was placed on crafting subscriptions with similar interests, using a manual, static approach. However, with randomly created subscriptions, covering is improved, if only slightly, in 6 of the 11 brokers used. Most importantly the top three brokers, 1,2 and 3 all see reductions in

their root subscriptions. Root subscriptions are passed up the broker hierarchy towards top level brokers in KBNImpl operation so the fewer root subscriptions received by the top level brokers in the network is seen as an indication of how well subscriptions, below these top level nodes, are covering one another.

In conclusion, this worst case scenario, of randomly created subscriptions still results in increased, if only slight, covering across more than 50% of the brokers in the network. It is argued and predicted that with more realistic real-world data sets, or a wide-scale user study of publication and subscription generation, the data representing subscription covering would continue to show improvements.

6.5.3 Hop Count in Delivery

In this experiment dynamic clustering is evaluated. In this process subscribers subscribe to a randomly chosen broker with their individual subscription. The broker then forwards a subscriber information message (containing the subscriber's Medoid) to the policy server via the trigger broker. The policy server uses this to calculate where the subscriber should be placed and forwards this decision back to the subscriber's parent broker. It is subsequently the responsibility of this broker to notify the subscriber of its newly calculated matching broker, instructing the subscriber to move, the subscriber disconnecting and re-connecting.

In contrast to the approach taken by subscribers, each publisher calculates its Medoid and forwards this information directly to the policy server, via the trigger broker, before their first publication. The policy server advises the publisher as to where in the network they should publish based on their Medoid, addressed to the publishers UUID.

This experiment was run twice. In the first un-clustered (KBNImpl) experiment clients publish and subscribe to random brokers within the network. In the second clustered experiment (KBNCluster) the subscribers and publishers are placed by the policy server around brokers that match their interests. In both experiments the hop count is recorded for the delivery of the same set of publications to subscribers. Hop count in delivery is seen as key to the evaluation of KBNCluster. The number of hops between source publication and subscriber is a direct indicator of the accuracy of the placement of clients into clusters. Therefore this experiment was conducted to establish the effect of clustering on the number of hops taken to deliver messages across a broker network of eleven brokers.

Shown in Figure 34 in histogram format are the percentages of messages delivered across each number of hops for both a clustered (KBNCluster) and an un-clustered (KBNImpl) network. The data indicates that through the introduction of clustering, the number of hops over which a message passes when being delivered from source to destination is dramatically reduced. Shown

in the data is that the mean hop count in the clustered experiment being 2.06 hops, with a standard deviation of 1.80. For the un-clustered experiment the mean hop count was 3.99 hops with a standard deviation of 1.57. KBNCcluster has shown to reduce the mean number of hops, with the same set of subscription and publication data, by 50%.

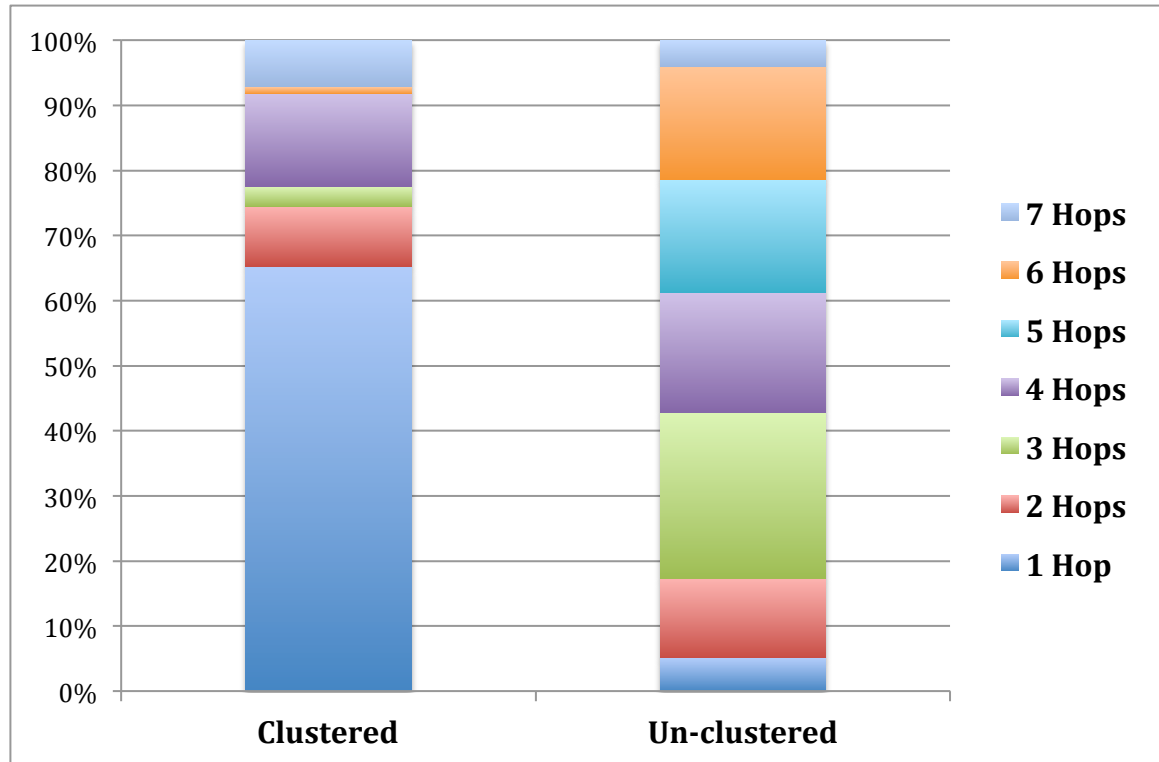


Figure 34: Hop Count - Clustered and Un-Clustered Topology

It is interesting to note that an anomaly occurs with regard to the publications requiring 6+7 hops (full traversal of the network). In the clustered experiment 8% of the messages are still delivered over 6 and 7 hops, while only ~4% require 6-7 hops in the un-clustered network. This data anomaly is attributed to the random nature of the creation of subscriptions and publications across the ontology and their potential placement in polar opposite clusters across the network. Medoids are calculated from the semantic subscriptions and publications of their clients. If the policy server receives a subscription with ontological attributes that are all extremely distant from one another and not particularly suitable, then the client may be placed in a cluster that results in a large number of hops for matching publications, hence the disparity in results.

This experiment has shown that KBNCcluster reduces the number of hops taken in routing messages from publisher to subscriber in when compared to KBNImpl. Hop count has been used in this thesis as a major characteristic of the performance of the KBNImpl, when compared against the total number of brokers in the network. This experiment has shown that a clustered KBNImpl, KBNCcluster, reduces the number of hops, across the whole network, taken in routing publications to subscribers when compared to KBNImpl. It is important to note that the hop count metric is only presented using a client-first approach to clustering. The high degradation of

performance associated with management messages requests, as shown in section 6.4.2 have been negated by having clients calculate their own semantic interests and forward these to the policy server. The hop count data is valid for a client first approach where the load placed on management is spread out to the edges of the network, the clients, as opposed to being placed directly upon the brokers.

6.5.4 Re-Clustering

The previous section has indicated that the approach taken in KBNCluster to the dynamic placement of clients into clusters substantially improves performance in terms of publication delivery hop count. However, there still remain a small number of clients that would benefit from re-clustering.

This experiment was conducted to explore the behaviour when, as clients' interests change, or if they are identified as being resident in a misplaced cluster, they are re-clustered by the policy server. This experiment is designed to show the impact of a policy to re-cluster clients if the hop count they experience in average message delivery was outside a specified range.

The policy server collects and monitors the mean hop counts required to deliver publications to individual subscribers from subscribers. This monitoring process allows for decisions to be made as to whether a subscriber has been misplaced into a cluster. If identified as residing in a sub-optimal cluster, the policy server re-examines the Medoid of the subscriber, searching for a more suitable cluster. The initial steps involved in this process are as follows:

1. Subscribers are initially clustered around brokers.
2. Publishers are initially clustered around brokers.
3. Notifications are delivered to subscribers, across the broker network.
4. Subscribers periodically inform the policy server, based on a timer activated policy, of the mean number of hops taken to deliver their notifications and the standard deviation of this mean value.

Step 4, above allows the policy server to identify that a subscribing client is in a sub-optimal cluster. However before the client can be re-clustered a client's subscription (and thus their Medoid) must reflect any change in their interests.

In KBNCluster if no change has occurred in their subscription then the client will be relocated back into the original cluster into which it resided. Until a subscriber's subscription changes, their cluster will not change. If however the client Medoid changes, after being identified as being in a sub-optimal cluster, the following occurs on the policy server:

1. Recalculation of the client's placement into a new cluster based on their new Medoid.
2. Continual monitoring of subscribers with excessive hop counts in notifications.

As the process for identification of clients in the wrong cluster reoccurs at scheduled intervals, it is assured that clients (identified as being in a sub-optimal cluster) will be re-clustered at some point in the future when their Medoid changes. With regard to publishers, it is the responsibility of publishing clients (after their initial clustering) to indicate to the policy server a change of interests and push to the policy server their new Medoid as a basis for being re-clustered.

In Figure 35 the mean of each subscriber's delivered notifications are plotted. From this data we can conclude that, with regard clustering:

- 71% of subscribers had messages delivered from a mean of between 1 to 2 hops.
- 14.5% of subscribers had messages delivered from a mean of between 2 to 4 hops.
- 14.5% of subscribers had messages delivered from a mean of between 6 to 7 hops.

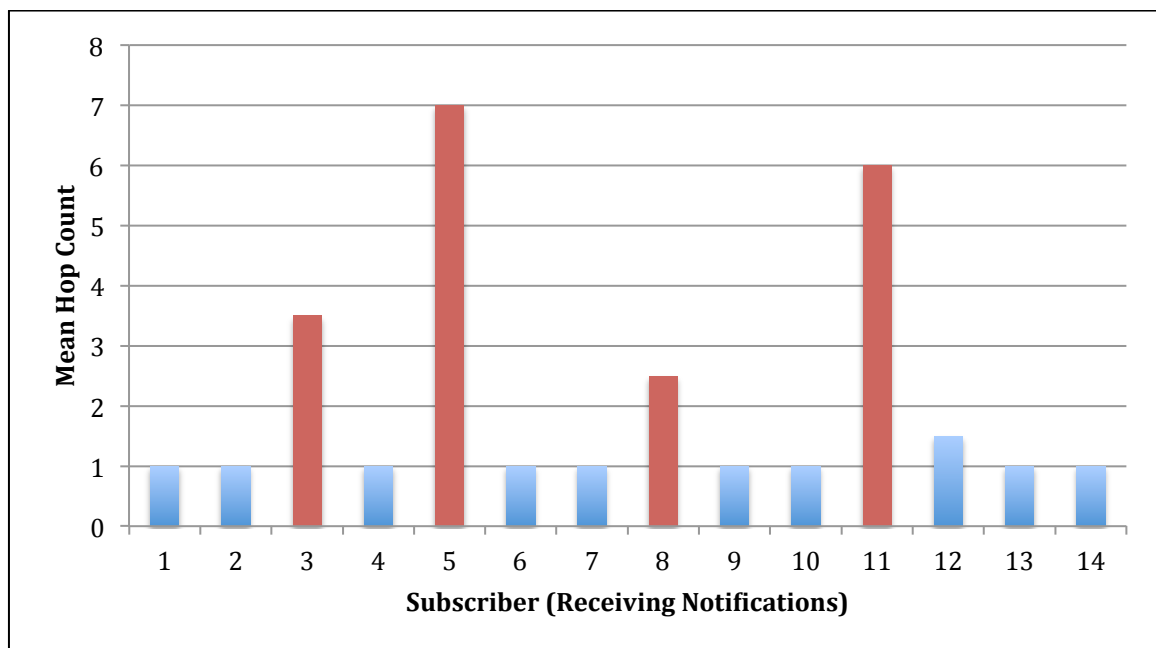


Figure 35: Mean Hop Count in Delivery for Each Subscriber

With regard to re-clustering, the data suggested that four of the subscribers were placed in clusters that possibly do not suit their interests (subscribers 3,5,8,11). In KBNCluster a policy rule determines that anything higher than 1.5 mean hops, for the given topology of eleven brokers, is too high. Given the data set plotted in Figure 35 we can discern that subscribers 3, 5, 8 and 11 are misplaced. Shown in Code Example 23 are the messages received by the four subscribers, instructing them that the policy server has determined they may be in the wrong cluster and that they should re-submit their subscriber information to the policy server for a possible re-placement. Should a subscriber receive a re-cluster message, as shown in Code Example 23, the subscriber re-submits their Medoid to the policy server. The policy server would then decide to re-cluster the subscriber to establish if the subscriber should reside in another more suitable cluster.


```
(UUID="TC:b562dd72.c.(mahler.cs.tcd.ie/127.0.1.1)1977220433" reCluster=true)
(UUID="TC:b562dd72.69.(mahler.cs.tcd.ie/127.0.1.1)1109376982" reCluster=true)
(UUID="TC:b562dd72.77.(mahler.cs.tcd.ie/127.0.1.1)1320864185" reCluster=true)
(UUID="TC:b562dd72.3f.(mahler.cs.tcd.ie/127.0.1.1)161514210" reCluster=true)
```

Code Example 23: Re-cluster Messages received by Subscribers

This experiment has shown in Figure 35 and Code Example 23 that subscribers have been identified as residing in a sub-optimal cluster based on policy rule, and that these subscribers have been notified to request re-clustering.

6.5.5 Conclusion

In conclusion this section has shown that the subscription tree size of brokers can be reduced, somewhat when brokers, publishers and subscribers are clustered in KBNCluster. In addition to this it has also been shown that clustering reduces the number of hops taken to deliver notifications. Finally it has been shown that the introduction of clustering has been extended to include the identification of clients, through usage reports, residing in a sub-optimal cluster and their subsequent, re-clustering.

6.6 Overall Conclusion

In this chapter six sets of experiments were introduced. Each experiment builds upon the previous in order to address the research question and objectives of this thesis. In this final section, the main findings from each experiment are outlined.

In Section 6.2, with regard to motivating static clustering, it has been demonstrated that a static approach to clustering supported continued investigation into the effect that clustering has upon KBNImpl, motivating the evaluation conducted as part of the rest of this chapter.

In Section 6.3, with regard to the evaluation of KBNImpl operational costs, it has been shown that different ontological operators have different costs associated with their use and that larger subscription trees take longer to search, particularly when using semantic subscriptions. Thus when clustered, the number of hops taken to deliver publications is reduced.

In Section 6.4.1, with regard to data storage and policy execution costs, it has been shown that the amount of memory and upper bounds of MIB updates / inserts updates well, in comparison to the experimental bounds placed upon KBNCluster (as shown in Section 6.1.3.) It is also shown that both the time taken to execute an increasing number of policies against MOs and the difference in time taken to execute various different policies scales well, again when compared to experimentation bounds placed upon KBNCluster.

In Section 6.4.2, with regard to data collection costs, it has been shown that overall performance decreases, in terms of subscription processing times, publication processing times and publication to subscription delivery times, as the rate of requests for management information increases. From this data a client first methodology to clustering subscribers and publishers, around a relatively static broker network, is motivated, as discussed in Section 6.4.2.7.

In Section 6.4.3, with regard to client mobility, it has been shown that it is more efficient to move a broker and all its attached subscribers than it is to move each individual subscriber from a broker. However the increased cost of moving individual subscribers, is offset against the cost of collecting management data from brokers, as discussed in Section 6.4.2. This finding further justifies KBNCluster using a client-first approach to clustering.

In Section 6.5, with regard to an overall evaluation of KBNCluster, it has been shown that the root subscription tree of most brokers (6 out of 11) has been reduced when a dynamic approach to clustering is taken. Although not as pronounced as possible, this is attributed to the random creation of subscriptions and publications. However, offsetting this, in the final evaluation conducted into hop counts, it has been shown that through clustering the hop count taken in message delivery is dramatically reduced when KBNCluster is compared to KBNImpl side-by-

side and importantly that clients, residing in a sub-optimal cluster, are identified and asked to re-cluster.

In the previous section 6.5 a fully dynamic client-first clustered topology is compared to a completely un-clustered and un-constrained broker network. In the last section of evaluation a clustered topology is compared to an un-clustered topology in terms of subscription tree size, hop count and the identification of clients requiring re-clustering. The data collection costs are not evaluated as the approach of management message requests and brokers satisfying these requests was dropped in favour of a client first approach to management data collection. In such a topology no additional load is placed on the broker networks, clients share the cost of management message calculation.

The **concluding chapter** readdresses the evaluation findings in terms of the research question and objectives of this thesis. From this discussion it is argued that managed dynamic clustering does improve the performance of Semantic-based Publish/Subscribe.

7 CONCLUSION

This thesis has presented and evaluated an approach to the clustering of Knowledge Based Network's clients and brokers as the prototype system that is KBNCluster. By utilising the semantics available through the use of KBNImpl, clusters of interest have been formed using a managed approach. In defining the semantic centre (Medoid) of a KBNImpl client and placing these clients around KBNImpl brokers routing publications from source to destination, KBNCluster is implemented. This clustering results in messages being delivered over fewer hops without management requests than when clients are un-constrained in their placement across the network, as is the case in KBNImpl.

This final chapter of this thesis presents an overview of the design and evaluation against the research question and objectives. Contributions of the research are discussed and future work is suggested as well as the achievements discussed.

7.1 Objectives and Achievements

The question this research set out to address was *whether a managed dynamic clustering approach improves the performance of Knowledge-based Networks, a sub-class of Semantic Publish/Subscribe?* From this, three main research objectives were drawn. These objectives are as follows:

Objective 1: Establish an approach for the formation, movement and re-clustering of semantic clusters in Knowledge Based Networks.

Objective 2: Establish the effect and overhead of implementing static and managed dynamic clustering in Knowledge Based Networks.

Objective 3: Apply Policy-based Network Management as an adaptable approach to the management of clustering.

In this section each research objective is broken down into a number of sub-objectives where necessary, and discusses how each was addressed in this thesis. For each research objective, supporting evaluation is identified from the previous evaluation chapter.

7.1.1 Research Objective 1

In addressing research objective 1, which was to “*Establish an approach for the formation, movement and re-clustering of semantic clusters in Knowledge Based Networks.*” three sub-objectives combine in part of completing this objective, these are:

Sub-objective 1: *Develop an approach for the formation of clusters.*

It has been shown in KBNCluster that publishers and subscribers residing in clusters appropriate to their interests receive messages with a lower hop count. Such a reduction in hop count is documented in the Evaluation Chapter 6, Section 6.5 (**Dynamic Clustering Evaluation.**)

Sub-objective 2: *Develop an approach for the movement of clients and brokers.*

The ability to move clients and brokers around the hierarchical topology of KBNCluster allows for clustering decisions to be enforced in clients. In addressing this sub-objective the broker, publisher and subscriber have been adapted enabling mobility across the network when requested to do so. The adaption was achieved via the embedding of a management agent into the brokers, publishers and subscribers themselves that enables management instructions to be sent to them by a manager system, i.e. the policy server. Each movement method was fully tested to ensure KBNCluster functioned as expected in KBNImpl i.e. publications that are delivered to subscribers. Each of these movement methods was evaluated in terms of the time taken to implement movement as shown in Chapter 6, Section 6.4.3 (**Mobility Costs**). From this data it can be concluded that the different methods of movement have different associated costs - the most expensive being moving all subscribers, individually, from broker to broker, and the cheapest being moving a broker and all attached subscribers in union across the broker network.

Sub-objective 3: *Develop an approach for identifying clients that should be re-clustered and a means by which that re-clustering could occur.*

The clustering mechanism was designed so clusters are formed and clients assigned to clusters in an automated dynamic manner. Clients are identified as residing in a sub-optimal cluster if the hop count associated with delivery of their publications passes a pre-determined configurable threshold of hops. It is implied that if a hop count, for a delivered notification, is determined to be too high then there may be a cluster which is more suited to the subscriber and the subscriber is re-clustered, if appropriate, based on their current semantic interests. An evaluation of the process of both the identification of and subsequent re-clustering of subscribing clients residing in an unsuitable cluster is discussed in Chapter 6, Section 6.5.4 (**Dynamic Clustering – Re-clustering**)

7.1.2 Research Objective 2

This second objective was addressed by *establishing the effect and overhead of implementing static and managed dynamic clustering in Knowledge Based Networks*. An initial evaluation studied the effect that static semantic clustering had on brokers, publishers and subscribers in KBNImpl. The evaluation examined the performance benefits offered to the clients and brokers through clustering.

Achieving this objective required determining the measures to be used in evaluating the performance of the subsequently implemented KBNCluster. The following metrics were established as having a direct effect on the performance of KBNCluster and were therefore central to the evaluation of this thesis:

1. *The size and make up of the subscription set held on message brokers.*
2. *The hop count taken in delivering publications to interested subscribers representing the distance taken in delivering messages across the network.*

Each of the above was evaluated in detail in a static topology, using KBNImpl, in Chapter 6, Section 6.2 (**Static Approach to Clustering**), and using KBNCluster in Chapter 6, Section 6.5 (**Dynamic Clustering Evaluation**).

7.1.3 Research Objective 3

In addressing research objective 3, which aimed to *investigate policy-based management as an adaptable approach to the management of clustering*, three metrics were evaluated, these are

1. The cost of collecting management data from across the managed overlay.
2. The cost of storing management data on the policy server.
3. The cost of evaluating firing policy rules against stored management data.

The evaluation conducted as part of this objective investigated the effect that the management of that clustering process had upon the performance of KBNCluster. It has been shown that, through the combination of the trigger broker, policy server, data collection methodology, data storage and policy execution process a flexible dynamic management approach for the grouping of clients into clusters is provided in KBNCluster. This evaluation is presented in Chapter 6, Section 6.4.1 (**Management Data Storage and Policy Execution Costs**).

In particular, it has been shown that clustering reduces the number of brokers required to route each message, and thus the broker network as a whole, when routing messages from publishers to subscribers. From this evaluation, it is concluded that KBNCluster is well suited to semantic clustering, despite the slightly increased costs associated with the management of this clustering when these costs are considered, offset by gains in performance associated with clustering.

7.2 Contributions

This research makes two novel contributions to the field of semantic based publish/subscribe. In this section these will each be discussed in turn.

7.2.1 Major Contribution

The major contribution of this work is:

- *A method for the semantic clustering of publishers, subscribers and brokers in a KBN.*

This contribution has been implemented in a prototype, KBNCluster, as part of this thesis and provides a platform for the clustering of KBNImpl, which has been shown to reduce the hop count required to deliver publications to subscribers and thus decrease the time taken to deliver publications.

Novelty: In KBNCluster, clustering is achieved by transposing an ontology onto an **A*** variation of the Dijkstra [47] graph of weighted nodes and edges. This graph is then used to calculate a single point in the ontology, from a source set of query interests returning, as a client Medoid, which is in the author's knowledge, both novel and new. Additionally being able to form clusters around a source ontology and placing clients into suitable clusters is possible through the introduction of the Medoid, as implemented in KBNCluster. **Impact:** The approach taken to clustering in this thesis has shown that semantics provide an important external model of reference, allowing clustering to be realised in SBPS as is in topic-based, re-igniting the possibilities of clustering Semantic-based Publish/Subscribe.

This major contribution is supported in the following publications:

- John Keeney, Dominic Jones, Dominik Roblek, David Lewis, and D. O'Sullivan, **"Knowledge-based Semantic Clustering"**: conference paper presented at The 23rd Annual ACM Symposium on Applied Computing (SAC 2008), Fortaleza, Ceará, Brazil, March, 2008.
 - This conference paper presented and evaluated a static approach to clustering, where each client was manually configured with a pre-defined cluster. This work supported the arguments behind the benefit to Knowledge-based Networks of clustering.
- John Keeney, Dominic Jones, Dominik Roblek, David Lewis, and D. O'Sullivan, **"Improving Scalability in Pub-Sub Knowledge-Based Networking by Semantic Clustering"**: conference paper presented at the 6th International Conference on Ontologies, DataBases, and Applications of Semantics, Algarve, Portugal, Nov 2007.

- Work from this thesis helped form the arguments for the motivation of clustering KBNs in this paper.

7.2.2 Minor Contribution

The minor contribution of this thesis' is:

- *An efficient approach for the management of KBNCluster.*

The combination of data collection, MIB design, the processes of populating and of updating MIBs, policy execution and policy enforcement, represent a novel approach to the management and clustering of KBNs. The union of this management approach with the use of the KBNImpl-based trigger broker allowing for management clustering actions to be enforced across the network.

Novelty: Using Semantic-based Publish/Subscribe for the collection of management data provides a set of triggers that are applied across a wide set of source data in the form of publications and subscriptions. The design of the MIB and policy execution across the MIB follows network management practice. However the way in which data is delivered to the policy engine is unique in KBNCluster. Using Semantic-based Publish/Subscribe for management data collection allows managed nodes to become publishers and system administrators (designing policy rules) to become subscribers, the two combining via a trigger broker and Semantic-based Publish/Subscribe for management message event delivery.

Impact: Where many management entities are producing content and only a few interested in consuming a sub-set of this content Semantic-based Publish/Subscribe, as used in the management of KBNCluster, via the trigger broker, becomes useful for knowledge filtering. Such an approach is applicable in many areas outside of those discussed in this thesis where the use of Semantic-based Publish/Subscribe provides a new approach for push-based management data collection.

This minor contribution is supported in the following publications:

- Dominic Jones, John Keeney, David Lewis, and D. O'Sullivan, "**Knowledge Delivery Mechanism for Autonomic Overlay Network Management**": conference paper presented at the sixth International Conference on Autonomic Computing and Communications (ICAC 2009), Barcelona, Spain, June, 2009.
 - This paper presented the mechanism used in collecting management data from across the broker network. This was a novel approach to collecting management data developed as part of this thesis.
- Dominic Jones, John Keeney, David Lewis, and D. O'Sullivan, "**Policy-based management of Semantic Clustering**" conference paper presented at the 2nd

International Conference on Distributed Event-Based Systems (DEBS 2008), Rome, Italy, July 2008.

- This paper presented the policy-based approach used in this thesis for controlling the clustering of publishers, subscribers and brokers. The application of policy to publish/subscribe clustering is a novel contribution of this research.

7.2.3 Additional Supporting Publications:

This research has sought to present the contributions of this thesis to influence the State of the Art through a number of additional peer-reviewed scientific publications, presentations and an academic book chapter, not directly linked to any contribution but presented below:

- John Keeney, Dominic Jones, Song Guo, David Lewis, and D. O'Sullivan, **“KNOWLEDGE-BASED NETWORKING”**, book chapter: Published in the "Handbook of Research on Advanced Distributed Event-Based Systems, publish/subscribe and Message Filtering Technologies." IGI Global (Editor(s): Annika Hinze and Alejandro Buchmann) 2009.
 - This peer reviewed book chapter introduced and presented a complete overview of the Knowledge-based Network, used in implementing KBNCluster. Work from this thesis contributed the related work, motivational case studies and discussion/future work sections of this chapter.
- Dominic Jones, John Keeney, David Lewis, and D. O'Sullivan, **“Knowledge-based Networking”**: conference paper presented at the 2nd International Conference on Distributed Event-Based Systems (DEBS 2008), Rome, Italy, July 2008.
 - This demonstration paper presents the KBNImpl and full range of semantic operators, an extension of the Siena CBN.
- John Keeney, Dominik Roblek, Dominic Jones, David Lewis, and D. O'Sullivan, **“Extending Siena to support more expressive and flexible subscriptions”**: conference paper presented at the 2nd International Conference on Distributed Event-Based Systems (DEBS 2008), Rome, Italy, July 2008.
 - This paper discusses the extensions to the Siena CBN which are implemented as part of the KBNImpl. Work from this thesis has provided the motivational case studies section and future work for this paper.

7.3 Future Work

Many of the ideas and research challenges surrounding this thesis could be addressed in more depth in future work. Included below are suggestions for and discussion of the next steps required in exploring further the research presented in KBNCluster.

7.3.1 Subscriber Re-clustering

In the current approach, subscribers are re-clustered based on the identification of too high a number of hops in routing between source and destination. The policy server is sent information by the subscriber that contains:

1. The number of notifications received by the subscriber.
2. The average hop count based on all messages delivered to the subscribers.
3. The standard deviation associated with the reported average hop count value.

Once a client is identified as being in an unsuitable cluster, based on too high hop count associated with the delivery of messages, the policy server reassigns the subscriber, placing it in a more suitable cluster.

An enhancement of this approach creates a decentralised role for subscribers in the identification of a change in their interests and the subsequent determination that their cluster placement is possibly sub-optimal, where the client automatically requests replacement. Such an approach removes the requirement on the policy server to identify clients in the wrong cluster, placing the responsibility instead on the subscriber, further farming out the processing costs associated with clustering.

7.3.2 Load Balancing of Clusters

The manner, in which the clusters are currently created and then overlaid onto the brokers, ensures that the less specific clusters are placed at the top of the broker network, and the more specific clusters placed at the bottom of the broker network. This current methodology does not, however, address the issue of load, or popularity, once clusters are overlaid across the broker network, nor does it adapt the number of brokers assigned to each cluster, as load changes.

This current approach can be improved upon through the introduction of load balancing i.e. adjusting the number of brokers per cluster in keeping with the cluster's load. Load balancing represents the provision of resources for optimal scalability, in terms of available hardware resources, whereas clustering represents the placement of clients into areas of similar interest within the network overlay, optimising the number of hops in message delivery. In the current implementation, a cluster is represented by a fixed number of brokers, which can result in the broker for a popular cluster becoming overloaded in terms of hardware resources, as the number

of attached clients increases. An extension to the current approach would allow for the policy server to offer a load balancing service, adding and removing resources to clusters and cultivating clusters which contain multiple assigned brokers, as required. Note that the mechanism for such load balancing and its integration into the policy server is already implemented but not evaluated as it is seen as outside of the scope of the thesis research question and research objectives.

7.3.3 The trigger broker as a management event monitoring system

The trigger broker is currently used as a communication mechanism between nodes in the overlay (managed loads) and the policy server (manager). The use of a KBNImpl based trigger broker allows for a set of semantic subscriptions from a given manager to be matched against management data sourced from across the overlay, and this information to be delivered as notifications. Such an approach has been shown to work well for the process of clustering, as presented in this thesis. An investigation into the properties of the trigger broker, or a KBNImpl-based approach to management, in a push-based approach, is an area in which additional research effort is justified.

7.3.4 Multiple Policy Servers

KBNCluster is implemented using a single policy server, which has overall control of the managed overlay. Instructions are sent from the policy server to the managed overlay, containing the policy server's unique identifier (UUID), as a string value.

Although the current deployment consists of a single policy server, trigger broker and managed overlay, the design of the architecture was built around the principle that multiple policy servers could interact with either single or multiple managed overlays simultaneously. The trigger broker is responsible for routing incoming management messages to multiple managerial subscribers, each uniquely identified.

This could, in future work, provide for multiple policy servers, operating in parallel, to control a single managed overlay, with management messages being choreographed by the trigger broker.

7.4 Final Remarks

This thesis has presented an approach for the clustering of KBNImpl, in KBNCluster, so that subscribers and publishers reside around clusters of common interest. This clustering is achieved using a policy-based approach to the management of client placement, and subsequent identification of clients placed in un-suitable clusters and their re-clustering.

This research has presented an approach to KBN clustering, the background technology and terminology have been introduced, and the design and implementation of KBNCluster documented at each stage of development. An approach to the formation of clusters and movement and re-clustering of semantic clients has been developed and evaluated in terms of the effect and overhead of the clustering process on KBNImpl. Additionally policy-based management has been shown to be effective in providing a mechanism for controlling KBNCluster, where the evaluation presented as part of this research assesses the cost of both static and dynamic clustering, management data storage, policy execution and data collection as well as evaluating mobility costs.

The key metric used throughout this thesis has been the hop count taken in delivering a publication to a subscriber. It has been shown through evaluation that 60% of publications were delivered across 1-4 hops in an un-clustered implementation of KBNImpl. In KBNCluster the same data set was used but 91% of publications delivered across 1-4 hops, **clustering has shown to increase the percentage of publications delivered across 1-4 hops by 31%**. this is the key finding of the evaluation of KBNCluster. Other evaluation data supports the design and implementation of the approach taken, but this key finding supports the argument that KBNCluster brings together publishers and subscribers around brokers of common interests, semantic clusters. In the final section of evaluation, section 6.5, a full implementation of KBNCluster is compared to a fully un-clustered topology. This provides a comparison of the benefits of clustering when comparing KBNCluster to KBNImpl. This evaluation is conducted with a client-first approach to clustering where any costs involved in calculating a clients Medoid is shared amongst the edges of the network, the clients, as opposed to the brokers.

In conclusion, semantic clustering around ontologies has been shown to be beneficial, and its applicability is only just beginning to be explored in other areas of computing. Wherever there are a great number publishing clients and subscribing individuals, using a semantic reference model, clustering as presented in this thesis can bring those clients together across an overlay. The main effect of this is a reduction in overall traffic, as messages are routed to clients more closely connected to one another and messages propagating across a network in fewer hops. KBNCluster uses semantics at its core; this thesis has shown that that when available semantics are fully utilised, content is delivered more efficiently and with less wasted message routing.

Bibliography

- [1] TIBCO, "TIBCO Software Inc," 2000.
- [2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-area Event Notification Service (Siena)," *ACM Transactions on Computer Systems*, vol. 19, 2001.
- [3] G. Antoniou and F. Van Harmelen, "Web Ontology Language: Owl," *Handbook on Ontologies*, vol. 2, 2004.
- [4] S. Guo, "Using Semantic Mappings for Semantic Based Publish/Subscribe Systems," PhD Thesis, Computer Science and Statistics Trinity College Dublin, Dublin, Ireland, 2009.
- [5] D. Lewis, J. Keeney, D. O'Sullivan, and S. Guo, "Towards a Managed Extensible Control Plane for Knowledge-based Networking," *DSOM*, vol. 4269, 2006.
- [6] J. Keeney, D. Roblek, D. Jones, D. Lewis, and D. O'Sullivan, "Extending Siena to Support More Expressive and Flexible subscriptions," presented at the 2nd International Conference on Distributed Event-Based Systems, Rome, Italy, 2008.
- [7] J. Keeney, D. Jones, D. Roblek, D. Lewis, and D. O'Sullivan, "Knowledge-based Semantic Clustering," presented at the 23rd Annual ACM Symposium on Applied Computing, Fortaleza, Ceara, Brazil, 2008.
- [8] D. Jones, J. Keeney, D. Lewis, and D. O'Sullivan, "Knowledge-based Networks," presented at the 2nd International Conference on Distributed Event-based Systems (DEBS08), Rome, Italy, 2008.
- [9] J. Keeney, D. Jones, S. Guo, D. Lewis, and D. O'Sullivan, "Knowledge-based Networking," in *The Handbook of Research on Advanced Distributed Event-Based Systems, Publish/Subscribe and Message Filtering Technologies*, ed: IGI Global, 2009.
- [10] M. Castro, P. Druschel, A. M. Kermarrec, and A. I. T. Rowstron, "Scribe: a Large-scale and Decentralized Application-level Multicast Infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, 2002.
- [11] G. Muhl, L. Fiege, and A. Buchmann, "Filter Similarities in Content-Based Publish/Subscribe Systems," in *Trends in Network and Pervasive Computing*, ed: LNCS, 2002.
- [12] T. Milo, T. Zur, and V. Elad, "Boosting Topic-based Publish/Subscribe Systems with Dynamic Clustering," presented at the ACM SIGMOD International Conference on Management of Data, Beijing, China, 2007.
- [13] S. Baehni, P. Eugster, and R. Guerraoui, "Data-aware Multicast," presented at the 2004 International Conference on Dependable Systems and Networks, Florence, Italy, 2004.
- [14] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "TERA: Topic-based Event Routing for peer-to-peer Architectures," presented at the Inaugural International Conference on Distributed Event-Based Systems (DEBS), Toronto, Ontario, Canada, 2007.
- [15] C. Gregory, M. Roie, T. Yoav, and V. Roman, "SpiderCast: a Scalable Interest-aware Overlay for Topic-based Pub/Sub Communication," presented at the Inaugural International Conference on Distributed Event-based Systems, Toronto, Ontario, Canada, 2007.
- [16] L. Querzoni, "Interest Clustering Techniques for Efficient Event Routing in Large-scale Settings," presented at the 2nd International Conference on Distributed Event-based Systems, Rome, Italy, 2008.
- [17] A. Wun and H. A. Jacobsen, "A Policy Management Framework for Content-based Publish/Subscribe Middleware," presented at the International Conference on Middleware, Newport Beach, CA, USA, 2007.
- [18] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody, "Role-based Access Control for Publish/Subscribe Middleware Architectures," presented at the 2nd International Workshop on Distributed Event-based Systems, San Diego, California, 2003.

- [19] R. Boutaba and I. Aib, "Policy-based Management: A Historical Perspective," *The Journal of Network & Systems Management*, vol. 15, 2007.
- [20] G. Klyne, J. Carroll, and B. McBride, "Resource description framework (RDF): Concepts and Abstract Syntax," *W3C Recommendation*, vol. 10, 2004.
- [21] D. McGuinness and F. Van Harmelen, "OWL Web Ontology Language Overview," *W3C Recommendation*, vol. 10, 2004.
- [22] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, *The Description Logic Handbook*, 2007.
- [23] J. Grossnickle, T. Board, B. Pickens, and M. Belmont, "RSS - Crossing Into the Mainstream " Yahoo! & Ipsos Insight, 2005.
- [24] G. Muhl, F. Fiege, and P. Pietzuch, *Distributed Event-Based Systems*, 2006.
- [25] P. Pietzuch and J. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," presented at the 22nd International Conference on Distributed Computing Systems, 2002.
- [26] M. Astley, J. Auerbach, S. Bhola, G. Buttner, M. Kaplan, K. Miller, R. Saccone Jr, R. Strom, D. C. Sturman, and M. J. Ward, "Achieving Scalability and Throughput in a Publish/Subscribe System," 2004.
- [27] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, "Content-based Routing with Elvin4," presented at the AUUG2K, Canberra, Australia, 2000.
- [28] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito, "Efficient Publish Subscribe through a Self-Organizing Broker Overlay and its Application to SIENA," *The Computer Journal* vol. 50, 2007.
- [29] J. Wang, B. Jin, and J. Li, "An Ontology-Based Publish/Subscribe System," presented at the International Middleware Conference, Toronto, Canada, 2004.
- [30] P. Milenko, I. Burcea, and H.-A. Jacobsen, "S-ToPSS: Semantic Toronto Publish/Subscribe System," presented at the 29th International Conference on Very large Data Bases, Berlin, Germany, 2003.
- [31] D. Roblek, "Decentralized Discovery and Execution for Composite Semantic Web Services," M.Sc. Thesis, Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland, 2006.
- [32] J. Martin, "Policy-based Networks," 1999.
- [33] M. Uschold, P. Clark, F. Dickey, C. Fung, S. Smith, S. Uczekaj, M. Wilke, S. Bechhofer, and I. Horrocks, "A Semantic Infosphere," *DSOM*, 2003.
- [34] M. Petrovic, H. Liu, and H. Jacobsen, "G-ToPSS: fast filtering of graph-based metadata," presented at the World Wide Web Conference, Chiba, Japan, 2005.
- [35] H. Li and G. Jiang, "Semantic Message Oriented Middleware for Publish/Subscribe Networks," presented at the Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense III, Orlando, FL, USA, 2004.
- [36] ("W3C Recommendation: OWL Web Ontology Language Overview", Accessed October 2011). <http://www.w3.org/TR/owl-features/>.
- [37] P. Alex, R. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl, "Designing Semantic Publish/Subscribe Networks using Super-Peers," in *Semantic Web and PeerTo-Peer*, ed: Springer, 2004.
- [38] M.-J. Park and C.-W. Chung, "iBroker: An Intelligent Broker for Ontology Based Publish/Subscribe Systems," presented at the International Conference on Data Engineering, 2009.
- [39] E. Prud'Hommeaux and A. Seaborne, "SPARQL query language for RDF," *W3C Working Draft*, vol. 4, 2006.
- [40] S. Voulgaris, E. Riviere, A. M. Kermarrec, and M. van Steen, "Sub-2-sub: Self-organizing Content-based Publish/Subscribe for Dynamic and Large Scale Collaborative Networks," presented at the 5th International Workshop on Peer-to-Peer Systems, 2006.
- [41] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *Lecture Notes in Computer Science*, vol. 2218, 2001.

- [42] R. Meier and V. Cahill, "Taxonomy of Distributed Event-Based Programming Systems," presented at the 22nd International Conference on Distributed Computing Systems, Montreal, Canada, 2002.
- [43] G. Flouris, D. Plexousakis, and G. Antoniou, "A classification of ontology change," presented at the 3rd Semantic Web Applications and Perspectives Workshop, Pisa, Italy, 2006.
- [44] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*, 1990.
- [45] M. van der Laan, K. Pollard, and J. Bryan, "A new Partitioning around Medoids Algorithm," *Journal of Statistical Computation and Simulation*, vol. 73, 2003.
- [46] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, 1968.
- [47] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, 1959.
- [48] T. Pedersen, S. Patwardhan, and J. Michelizzi, "WordNet:Similarity: measuring the relatedness of concepts," presented at the ACL Workshop on Empirical Modeling of Semantic Equivalence and Entailment, Arbor, Michigan, USA, 2005.
- [49] M. Proctor, M. Neale, P. Lin, and M. Frandsen, "Drools documentation," *JBoss*, vol. 5, 2008.
- [50] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: Implementing the Semantic Web Recommendations," 2004.
- [51] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A Practical Owl-dl Reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, 2007.
- [52] ("The Protégé Ontology Editor and Knowledge Acquisition System", Accessed October 2011). <http://protege.stanford.edu/>.
- [53] ("Standard Error Calculation", Accessed April 2013). <http://davidmlane.com/hyperstat/A103735.html>.
- [54] B. Douglass, "UML Sequence Diagrams," *Embedded Systems Programming*, vol. 16, 2003.
- [55] K. Burtch, *Linux shell scripting with Bash*: Pearson Higher Education, 2004.
- [56] ("Ontology Alignment Evaluation Initiative 2007", Accessed October 2011). <http://oaei.ontologymatching.org/2007/>.
- [57] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Waerzoniak, and M. Bownman, "PlanetLab: an overlay testbed for broad-coverage services," *SIGCOMM Computer Communication Review*, vol. 33, 2003.
- [58] J. Keeney, D. Lewis, and D. O'Sullivan, "Benchmarking Knowledge-based Context Delivery Systems," presented at the International Conference on Autonomic and Autonomous Systems, Silicon Valley, CA, USA, 2006.

Appendices

Appendix A

John Keeney, Dominic Jones, Dominik Roblek, David Lewis, and D. O'Sullivan, "Knowledge-based Semantic Clustering", Pages 460-467, presented at The 23rd Annual ACM Symposium on Applied Computing (SAC 2008), Fortaleza, Ceará, Brazil, March, 2008.

Knowledge-based Semantic Clustering

John Keeney, Dominic Jones, Dominik Roblek, David Lewis, Declan O'Sullivan

Knowledge & Data Engineering Group (KDEG) & CTVR

Trinity College Dublin, Ireland

{ John.Keeney | jonesdh | roblekd | Dave.Lewis | Declan.OSullivan }@cs.tcd.ie

ABSTRACT

Users of the web are increasingly interested in tracking the appearance of new postings rather than locating existing knowledge. Coupled with this is the emergence of the Web 2.0 movement (where everyone effectively publishes and subscribes), and the concept of the "Internet of Things". These trends bring into sharp focus the need for efficient distribution of information. However to date there has been few examples of applying ontology-based techniques to achieve this. Knowledge-based networking (KBN) involves the forwarding of messages across a network based not just on the contents of the messages but also on the semantics of the associated metadata. In this paper we examine the scalability problems of such a network that would meet the needs of Internet-scale semantic-based event feeds. This examination is conducted by evaluating an implemented extension to an existing pub-sub content-based networking (CBN) algorithm to support matching of notification messages to client subscription filters using ontology-based reasoning. We also demonstrate how the clustering of ontologies leads to increased efficiencies in the subscription forwarding tables used, which in turn results in increased scalability of the network.

Categories and Subject Descriptors

C.2.2 Network Protocols: Routing protocols, I.2.4 Knowledge Representation Formalisms and Methods: Semantic Networks

General Terms

Performance, Experimentation

Keywords

Publish-subscribe, content-based networking, ontologies

1 INTRODUCTION

Establishing a global event service at Internet scales presents a major challenge for existing networking technologies. Such an event service is crucial in the support of the explosion of dynamic interactivity expected through the increased use of Web 2.0 technologies where diverse and an increasing numbers of publishers and subscribers of content will be more mobile and dynamic [1]. The time at which items are posted is increasing in importance relative to the content of the post, e.g. blog postings rapidly fade in importance as time passes. The web has responded

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.
Copyright 2008 ACM 978-1-59593-753-7/08/0003...\$5.00.

to this need with RSS feeds which allow event postings, to quickly be notified to interested users. However, this system relies on users subscribing to feeds of pages they have already located, whilst feed aggregators offer only rudimentary searches or simple classifications of feeds. This is partly because the near-real time events present in feeds are disassociated from the system of user-defined hyperlinks required by search engines which also introduces a discovery latency that is unacceptable to feed users. Though we can search for the static pages we are unable to search the body of feed events active at any point in time. As we look forward to future uses of the Web, in support of the 'Internet of Things,' searching for events becomes more important as devices and sensors become sources of high frequency postings of interest. In [27] it is suggested that that an Internet-wide event service may need to scale to 10^9 events per second, a similar order of event producers and huge variability in the proportion of consumers subscribing to an event. Current event-based publish-subscribe systems offer a networking model that is well suited to such applications, but they are typically limited to relatively static characterisations of events. Elements of this are being addressed by developments in Content Based Networks (CBN), a specialisation of the pub-sub paradigm where message forwarding is based on message attributes and their values. Extensive research is ongoing into finding a balance between restricting the expressiveness of event attribute types and subscription filters, their efficient matching at CBN nodes and efficient maintenance of routing tables [11, 21, 22, 33]. Currently user subscriptions are limited to simple syntactic matches (typically integers, strings and Booleans). In [16, 29, 31] the concept of Knowledge Based Networking is introduced, as a semantically enhanced publish-subscribe model extending content-based networking (CBN). This novel integration of semantics within the pub-sub routers themselves allows messages to be matched to subscriptions based not only on their contents, but also their semantics. Producers of knowledge express the semantics of their available information based on an ontological representation of that information, and publish semantically enriched messages as required. Consumers express subscriptions upon that information as long-lived semantic queries, in response to which they continually receive suitable matching messages. A Knowledge-based Network (KBN) is therefore more flexible, open and reusable to new applications. However, the scalability of a KBN to Internet scale requires a routing mechanism that minimizes both the size of routing state held in KBN nodes and the overhead of ontological reasoning in nodes. To address this, [29] proposes the efficient partitioning of the routing space based on clustering related to the semantics of message contents, rather than grouping within the hierarchies of network addresses. In this paper we describe some empirical evaluation into the performance of semantic-based clustering within a deployed KBN using realistic distribution of subscriptions, notifications and their semantics based on characteristics of existing RSS feeds.

2 KBN IMPLEMENTATION

The particular flavour of KBN investigated in this paper is an extension of the Siena CBN [12]. A Siena notification is a set of typed attributes, each attribute is comprised of a name, a type and a value. The current version of Siena supports the following basic types: string, long, integer, double and Boolean. Siena subscriptions are a conjunction of filtering constraints, where constraint is comprised of the attribute name, an operator and a value. The subscription operators currently supported are equality and less/greater than etc., and for strings, substring, suffix and prefix. Each Siena router maintains its own set of subscriptions (routing table), which is dynamically built from the specific subscriptions it receives. A subscription “covers” a notification, if the event matches to all filtering constraints of a filter. Notifications are delivered to a client, if the client has submitted a subscription where the conjunction of the subscription’s filters covers that notification. Siena also discovers coverings between subscription filters to optimize subscription routing. A filter covers another filter, if all notifications covered by the latter are also covered by the former. The Siena covering relationships, defined in [12, 20], allow each router’s subscription set to be dynamically arranged into a hierarchical tree structure (routing table), with more general subscriptions towards the top, and more specific subscriptions towards the bottom. This structure allows subscriptions to be efficiently and correctly aggregated together to reduce the subscription tree size and efficiently match each publication to subscriptions as it passes through each router.

In [16, 29, 31] a KBN implementation is presented that extends Siena by providing three additional ontological base types: properties, concepts and individuals. It also supports subsumptive subscription operators, i.e. sub-class/property (more specific), super-class/property (less specific), and equivalence. E.g., a subscriber can subscribe to all KBN messages that contain an attribute whose value is a concept more/less specific than the named concept in the subscription. To achieve this, each KBN router holds a copy of a shared OWL ontology, within which each ontological class, property and individual used is described and reasoned upon. These new ontological types are first class KBN types, and can be used in any KBN subscription or notification, along-side the standard Siena types and operators. Due to the transitive nature of the sub-property/class and super-property/class properties, covering relationship for these operators were defined in [31], marinating Siena’s subscription aggregation efficiencies. A further fully-implemented extension, presented in [34], introduces a new “bag” type and associated bag operators. A bag (also called a multiset) is a set-like object. The bags of integers {1,2,3} and {2,1,3} are equivalent, but bags {1,1,2,3} and {1,2,3} differ. The bag of integers {1,1,2,3,4} is a super-bag of {2,4,3} and so on. A bag can contain any valid Siena values, including other bags. Bags are first order members of the Siena KBN type set so can appear in notifications, as well as in subscription filters.

The bag operators can also be combined with other Siena KBN operators to produce composite bag operators. The composite bag relation is a binary relation over bags composed of (i) another binary bag relation over bags and of (ii) a sub-relation over the bag elements. The bag of integers {1, 1, 2, 3, 4} is a super-bag of {2, 4, 3} using the default “equals” (==) sub-relation. The bag of integers {1, 2, 3} is an equal-bag of {2, 3, 4} using the “less than” (<) sub-relation. (i.e. for every element in the second bag, there

exists an element in the first bag that is less than the element, with no unused elements in either bag). A full description and logical proofs of KBN bags, and the simple and composite bag operators are outside of the scope of this paper, but are provided in [34].

These bag type and operator extensions greatly extend the expressiveness of the Siena KBN subscription mechanism, especially when combined with the ontological operators. Again, due to the transitive nature of the sub/super bag operators, when combined with the covering relationships for the other Siena KBN operators, covering relationships for the bag operators can also be defined [34]. This maintains the efficiencies of Siena, allowing a single homogenous KBN to scale to moderate sizes.

3 BENCHMARKING KBN PERFORMANCE

Many of the parameters that affect the performance of a KBN’s routing scheme are largely application specific. Therefore a KBN can only be evaluated through its use in supporting diverse applications in a variety of scenarios. A benchmark, specifically for KBNs, is presented in [25], based on a synthetic benchmark for Content-based Networks in [26]. It defines the set of parameters that must be defined before an application of a specified KBN can be evaluated in either a qualitative or quantitative manner. These are summarised below:

Message generation: publication rate; subscription rate; active / inactive subscriptions cycle durations.

Publication generation parameters: number of fields in publication; publishers’ ontologies (defined in terms of content, size, complexity, expressiveness, bushiness etc.); names of attributes in publications, which may be drawn from publishers’ ontologies; type of each attribute; value space for each attribute, which may be drawn from publishers’ ontologies.

Subscription generation parameters: number of subscriptions per subscriber; number of filters per subscription; subscribers’ ontologies (defined in terms of content, size, complexity, expressiveness, bushiness etc. and its similarity to the publishers’ ontologies); names of attributes used in each filter, which may be drawn from the publishers’ or subscribers’ ontologies; type of each attribute used in the filter; attribute values used in filters, which may be drawn from the publishers’ or subscribers’ ontologies; operators used in filters.

KBN routers’ ontologies (defined in terms of content, size, complexity, expressiveness, bushiness etc. and their similarity to each other, the publishers’ ontologies and the subscribers’ ontologies). Only once the parameters listed above have been made explicit for each application running on top of a KBN, the performance can then be effectively and accurately compared.

4 PODCASTING – A REAL WORLD-BASED PUB - SUB USAGE SCENARIOS

In order to undertake empirical evaluation into the performance of a KBN using the benchmark identified in section 3, it was necessary to identify realistic distributions of subscriptions, publications and their semantics. Despite the increasing adoption of semantic-based metadata within the Web 2.0 community, there remains few sources of information to define distributions of subscriptions, messages and their semantics for different applications. In order to identify some realistic benchmark values we examined the distribution of podcast update feeds.

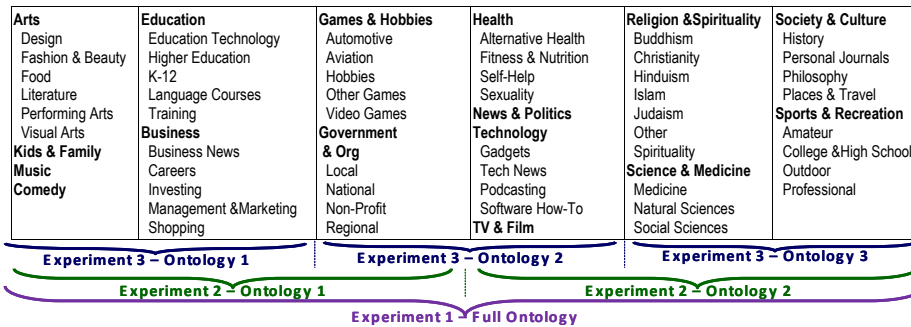


Figure 1: Categories in the Apple iTunes Podcast schema (Ontology)

4.1 PODCASTING

The application area chosen for study was podcasting, due to its popularity and the availability of a semantically rich de-facto metadata standard, the iTunes XML schema [7]. The iTunes schema allows for basic descriptors to be combined with semantically rich tags such as descriptive hierarchical categories, keywords, and ownership. Of particular interest is the set of hierarchical categories defined in the schema, as shown in Figure 1 which are used to annotate messages and aid in the searching for relevant podcasts. Firstly, it was necessary for the authors to establish exactly how many podcasts were actually being produced and consumed, or in terms of this paper, published and subscribed to. In 2006 a number of pivotal net analysts, including Neilson and Pew, aimed to answer this question. The survey by the "Pew Internet and American Life project" [3] in November 2006, through a telephone survey of 2,928 adults within the Continental US, showed 12% of Internet users had downloaded a podcast, averaging an estimated 65 million podcast listeners within the U.S.A. This is in contrast with the 7% of users who reported downloading a podcast in their April 2006 survey showing a growth of around 5% in as many months. The Neilson analysis [4] shows varying yet similar results to the data collected by Pew. Based on a phone interview of 1700 participants, 6% of the respondents were "regular podcast downloader's" leading to an estimate that 6% of US Adults, or around 9 million Web users had downloaded podcasts in the July-August period of 2006. Neilson also estimates that 72% of the respondents who regularly download podcasts download an average of 1-3 podcasts per week, 10% of whom download 8 or more podcasts per week. Neilson concludes approximately five podcast downloads per week was a fair estimate of average consumption. These studies provided us with a good indication of type of growth in podcast subscription that we should model in our distribution. From data donated by the administrators of FeedBurner.com (<http://www.feedburner.com>), one of the most popular and established podcast syndication websites, we determined the following with respect to characterising the distribution model for publications and subscriptions. In 2006, the number of podcast feeds grew from 31,167 feeds to 83,743 feeds, resulting in a growth of 52,576 new feeds over that 1 year period. When distributed equally over the year, this equates to a new podcast producer publishing approximately every 10 minutes. From a survey of the most 30 of the popular feeds, the average update period for each feed averaged one update per week. Where each

feed is represented by one publisher, and each publisher publishes a new notification every week, this means that a new publisher starts on average every 600 seconds, with an average continuous publication rate of one notification every 604,800 seconds per publisher. The data from FeedBurner.com also shows a growth in subscriptions from 915,277 to 6,434,758, resulting in 5,519,481 new subscriptions in 2006 alone. This can be approximated to 836,285 new subscribers over the year or one every 37.73 seconds. Based on the data in a Yahoo White paper on RSS feeds [9], each subscriber maintains an average of 6.6 subscriptions. It is estimated that podcast users very rarely change their subscriptions once they have found feeds that they like, and they rarely subscribe to feeds that they do not like. For these reasons this scenario estimates that each subscriber takes one week to subscribe to only their favourite 6.6 feeds, and then never unsubscribe. These approximations mean that a new subscriber is created on average every 37.73 seconds, which each creates an average of 6.6 subscriptions over a week (i.e. one every approx. 91,636 seconds or 25.5 hours), and then continuously waits for messages, the ratio shown in Figure 2:

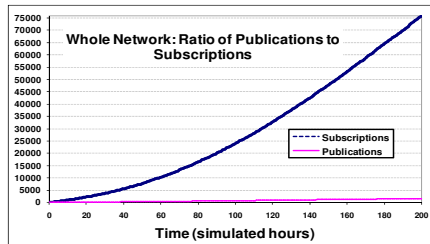


Figure 2: Ratio of Publications to Subscribers

4.2 DEPLOYING THE EXPERIMENT

In striving to replicate a real world distributed deployment, the PlanetLab network [10] was used to deploy a KBN network and exercise it according to the scenario above. The PlanetLab network comprises 756 nodes, distributed across 368 sites worldwide. Whereas traditional network simulations are evaluated on either a local or virtual test-bed, the PlanetLab environment allowed us to experiment across a physical Internet infrastructure of 77 random machines distributed across Europe, North America,

South America, Asia and Australia. The experimental setup consisted of 37 nodes running as dedicated KBN routers, 15 nodes running as dedicated publication creators and a further 25 were used as dedicated subscription generators. The 37 KBN routers form the hierarchical overlay as shown in Figure 3.

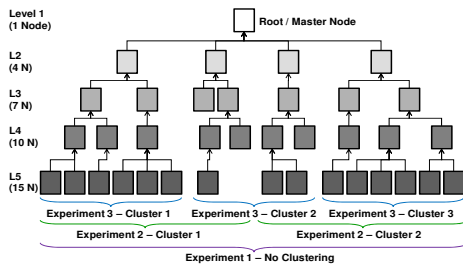


Figure 3: The KBN overlay network

4.3 THE PODCASTING BENCHMARK

To evaluate the performance of the KBN we simulated the distribution of podcast feed updates according to the traffic characteristics discussed in section 4.1. In this scenario a KBN publishing client is created for each feed, and the client generates KBN publications for every update announcing a new podcast episode for that feed. A KBN client was also created for each feed subscriber, and that client created a separate KBN subscription for each of its feed subscriptions. To speed up the gathering of data it was decided to speed-up the experiment by a factor of 365, i.e. model a full year's traffic in a single day.

Message generation rates: These are sourced from section 4.1: a new publisher was started on average every 600 seconds, with an average continuous publication rate of one notification every 604,800 seconds per publisher, a new subscriber was created on average every 37.73 seconds, which each created an average of 6.6 subscriptions over a week (i.e. one every approx. 91,636 seconds or 25.5 hours), and then continuously waited for messages.

Publication generation parameters: The publishers' relatively shallow and simple ontology was hand-crafted from the item categories as defined in the Apple iTunes podcast schema shown in Figure 1. This ontology is relatively small, at 38 kilobytes, with 67 classes, no properties and no individuals. Each publication message contained 15 named attributes, as defined in the Apple iTunes podcast schema. (*title, link, copyright, pubDate, itunes_Author, itunes_Block, itunes_image, itunes_duration, itunes_explicit, itunes_newFeedUrl, itunes_owner, itunes_subtitle, itunes_summary, itunes_category, itunes_keywords*). All of the named attributes except *itunes_category* and *itunes_keywords* were of type String. *itunes_category* was defined as a bag of ontological classes, and *itunes_keywords* was defined as a bag of Strings. Following a survey of the 30 most popular feeds hosted by FeedBurner.com, the average number of keywords in each podcast item/episode was calculated as 4, therefore the *itunes_keywords* bag of each publication contained 4 keyword strings. These keywords were randomly drawn from a dictionary of 80 popular keywords. In the same survey, an average of 3 categories were attached to each publication, so the *itunes_category* bag of each publication

contained on average 2-4 classes, drawn randomly from the publishers' ontology described above and shown in Figure 1. The values for all of the attributes except *itunes_category* and *itunes_keywords* were hard-coded as static strings.

Subscription generation parameters: Despite extensive searches we were unable to locate any information about how subscribers search for and select podcasts. For this reason, we decided to base the subscription format on what we considered the most useful and important semantic attributes of published podcast update notifications, i.e. the *itunes_keywords* and *itunes_category* attributes. When searching for actual podcasts the user would most likely use a search engine. The most popular search engines, including www.podcast-search.info, www.google.com, podcasts.yahoo.com and so on, all implement searches using a conjunction of keywords. In this scenario we decided to implement this subscription using the bag subscription mechanism discussed in section 2 by requiring that any matching subscription's *itunes_category* bag of keywords must be a *super-bag* of the keywords requested by the searcher. In this scenario, each subscriber subscribes using between 0 and 3 keywords randomly drawn from the same dictionary used by the publisher. When searching, a user would most likely select a single category, which would include all sub-categories if it was a parent category. Using the compound bag operator and described in section 2, this search can be implemented by requiring that the *itunes_category* attribute of any message must contain a bag of categories that is a super-bag of the required category, where one of the elements in the published *itunes_category* bag must be "more specific" than the required category, i.e. subsumed by the requested category. (i.e. a super-bag using the "more specific" sub-operator). In this scenario each subscriber specifies one category class drawn from the same ontology as the publisher. Therefore, each subscription is a conjunction of 2 constraints, 0-3 keywords and a category class. This is based on the experimental assumption that a user when searching for content is typically less specific than a user posting content. In this scenario only the *itunes_category* and *itunes_keywords* attributes were used in the subscription filters so it was acceptable to have the other unused attributes in the publications hard-coded as representative static strings.

Network topology: Combined with the goal of testing the KBN in a physically distributed environment, due to the resource requirements of this very large scale KBN deployment scenario (in particular the memory requirements of the multi-threaded publishing and subscribing clients rather than that of the routers), it was not possible to test the KBN's operation for this scenario without widely distributing the workload of the clients. As discussed in the section 4.2, the experiment was deployed across 77 distributed PlanetLab nodes, with 37 randomly selected nodes acting as KBN routers deployed in a hierarchy shown in figure 3, 15 randomly selected dedicated publishing nodes, and 25 randomly selected dedicated subscribing nodes. This workload distribution took into account the high subscription rate and relatively low publication rate. Considering the hierarchical nature of the network, and envisioning that the Root/Master node would suffer the highest loading, no publisher or subscriber sent messages directly to the Root Node.

KBN routers' ontologies: The KBN routers each used a copy of the same podcast categories ontology as used by the subscribers and the publishers. The hypothesis of this research is that the

clustering of KBN nodes according to the semantics of the knowledge they present or request will have a positive effect on the performance of the KBN and improve its scalability. To evaluate this hypothesis the operation of the KBN was evaluated in 3 experiments. The scenario described was evaluated by crudely dividing the KBN hierarchical overlay into clusters of approximately equal size, as shown in Figure 3. In the first experiment the network was not divided (one cluster). In the second experiment the same logical network hierarchy was divided into two clusters. In the third experiment the same network was divided into three clusters. When divided, each cluster was tasked with focusing on only a proportion of the ontology, as shown in Figure 1.

To demonstrate the expected difference in performance due to clustering, the KBN's operation was measured. In the first "unclustered" experiment, each publisher and subscriber could send their subscriptions and publications messages to any random node in the network (except the Root node). In the second and third "clustered" scenarios, depending on the semantics of the subscription or publication to be sent, the client selected a random node from whichever cluster was most suited to receive that message (i.e. the cluster that focussed on the portion of the ontology which contained the majority of their referenced ontological concepts). If the message referenced the same number of concepts from all portions of the ontology then the message could be sent to any node in the network. In each of the experiments the same volume of publications and subscriptions were created, according to the same message generation distributions described above. It is important to note however that although the cluster to receive the message was calculated, the particular node within the cluster that would receive the message was randomly selected from the cluster's members. This approach was taken to maintain the idea that in a hierarchical overlay pub-sub network, where the logical hierarchy may be very different from the physical network's topology, clients are not restricted by which router (or broker) they should connect to. In many cases clients may connect to their closest router, but this is not a requirement, hence random node selection could be considered a worst-case scenario.

5 RESULTS AND FINDINGS

The primary metrics used in this paper were to compare the performance of the KBN's operation where the characteristics of the subscription tree / routing table stored at each KBN router. This was due the end-to-end nature of the network, the heterogeneity of the machines in the PlanetLab network, and the variability of dynamic resource availability on the PlanetLab nodes as they ran other experiments concurrently. The authors feel that this is a fair metric to objectively compare the experiments for two reasons. Firstly, by the nature of using content-based subscription filter matching, for each notification potentially all subscriptions filters may need to be evaluated against the notification. For each subscription the subscription tree needs to be searched to find the optimal position to insert the subscription. Therefore, a smaller and more ordered subscription tree is more efficient. Secondly, the hierarchical logical topology of the KBN overlay were randomly created, and clients connect to random nodes (within a cluster), so the aggregate network traffic across each of the experiments should remain similar (especially for larger networks.)

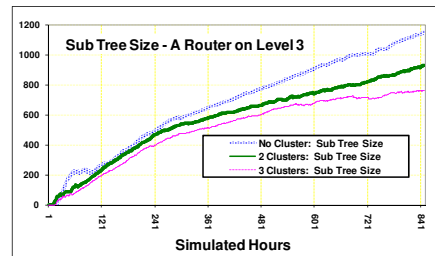


Figure 4: Subscription tree / Routing table size on one KBN router

Figure 4 shows the sizes of the subscription tree / routing table for a node at level 3 in the middle of the KBN hierarchy. The graphs are truncated to approximately 850 simulated hours since soon after this point the resources requirements of the subscribers began to exceed the conservative fair-use resource allocation of the oldest PlanetLab nodes, mainly due to the 365X speedup factor used in these experiments. Resource throttling at these weaker nodes meant that the results became unreliable after approx 1000 simulated hours (40+ simulated days). The graphs show how the total subscription tree size is smaller when semantic clustering is employed, despite a very similar number of subscriptions arriving at the node for each experiment. Despite the fact that the rate at which subscriptions were arriving continued to increase according to the distribution in figure 1, the total subscription tree size was starting to level out. The results therefore show that for a given number of received subscriptions, the subscription tree is smaller when semantic clustering is employed. Similar graphs were generated for nodes at each level in the hierarchy, and all show very similar results (except the Root node), and so are not presented here. The main optimisation feature of the Siena Hierarchical CBN, upon which the evaluated implementation of the KBN is based, is the use of subscription aggregation / covering to merge and order similar subscriptions. In the Siena subscription tree, subscriptions which cannot be merged with other subscriptions, or grouped under more general (covering) subscriptions, form "root" subscriptions in the tree. The count of "root" subscriptions, when considered with the size of the subscription tree, gives an overview of the searchability and optimality of the subscription tree at each node. Therefore the number of Siena "root" subscriptions at each node was also measured, as shown in Figures 5-8. The "root" subscriptions at each node are the most general subscriptions of each node so it is only these root subscriptions that need to be passed up to a router's parent node. To a parent router a child router appears as a subscribing client using only the child's "root" subscriptions.

This is done iteratively up the tree, so subscriptions become more general in the nodes towards the top of the network, and more specific towards the bottom. This explains why the more general subscriptions flowing up the hierarchy cause the subscription tree to reach optimality quicker as more general subscriptions reduce the number of "root" subscriptions. In addition, with respect to how child nodes send their "root" subscriptions to their parent node, it can be seen that semantic clustering greatly reduces this traffic and associated routing table "churn". This is particularly apparent towards the bottom of the network. Again similar graphs

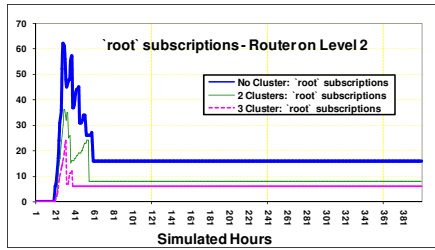


Figure 5: "Root" subscriptions in a level 2 router

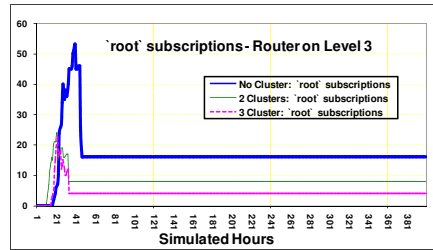


Figure 6: "Root" subscriptions in a level 3 router

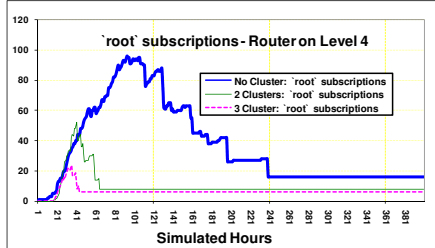


Figure 7: "Root" subscriptions in a level 4 router

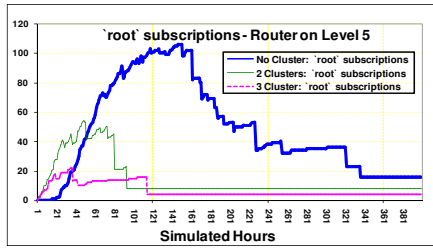


Figure 8: "Root" subscriptions in a level 5 router

were generated for other nodes at each level in the hierarchy, and all show similar results except the Root node. In a purely hierarchical network, where messages may be routed from one side of the network to the other side, it is clear that the Root/Master node can form a bottleneck in terms of the scalability of the network. As all the traffic travelling from one branch (and cluster) to another grows, the routing overhead in the Root node also grows. For this reason, to maximise scalability, it is necessary to minimise the size of the routing table at the root node, and optimise its searchability.

Figures 9 and 10 clearly show that for the application characteristics, KBN configurations, and scenarios introduced above, the subscription tree / routing table in the root node is reduced and converges to smaller size when even rudimentary semantic clustering is performed. In addition, since the subscribing clients do not send subscriptions directly to the Root node, the only subscriptions reaching the Root node come from the "root" subscriptions of the nodes on level 2 of the hierarchy. As a result of only receiving more general subscriptions, there is a much smaller difference between the total subscription tree size and the number of "root" subscriptions in the node. Shown in, specifically Figure 10, are the root subscriptions which are the covering subscriptions. The figures show the roots subs reaching maximum capacity, which with the introduction of new subscription content would begin a reversal of this trend.

6 RELATED WORK

There has been little examination of the use of ontology-based semantics in content-based networking in the scientific literature. In [17] a semantic publish-subscribe system is presented, but it is based on a centralized (pub-sub bus) implementation and thus is

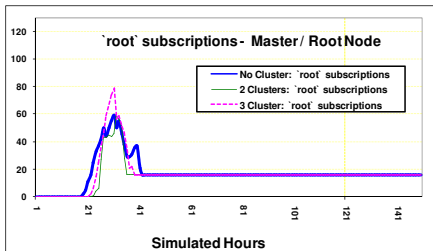


Figure 9: "Root" subscriptions in the Master/Root Node

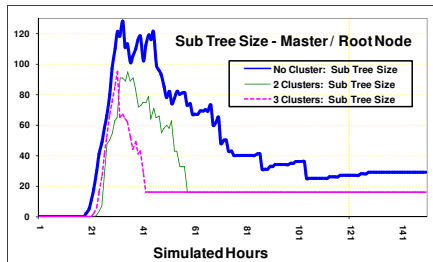


Figure 10: Subscription tree size in the Master/Root node.

limited to enterprise scale and does not offer true CBN capabilities. In [35], semantics can be used in messages in a pub-sub middleware; however, the semantics are used only at the edge of the network in a manner similar to a small scale study presented in [30]. The KBN presented in this paper uses semantics deep in the forwarding algorithm of each message router within the network. Much work to date on content-based networks has focused on how efficiency in routing can be gained through subscription aggregation and merging. Recent progress with the XNET CBN has shown that perfect routing can be achieved in a scaleable manner independently of subscriber joins and leave rates though subscription aggregation [33]. The HERMES CBN [22], TopSS [18] and the REBECCA CBN [24] have all applied peer to peer distributed hash table (P2P DHT) mechanisms to the formation of routing tables in CBN nodes. This is interesting in that it may form the basis for a flexible and robust clustering mechanism for routing in KBNs. It should be noted that though P2P systems themselves are concerned with efficiently routing queries to matching information sources in a query-response manner, they do not address the CBN concern of optimally routing a sequence of asynchronous replies back to the set of querying, or in CBN terms, subscribing clients. P2P DHTs provide efficient routing by using a cost metric keyed to the physical topology of the network resulting in average hop-counts for a route in the order of the log of the number of nodes in the network i.e. $O(\log(N))$. However a difficulty remains in the mapping of content based subs to a key space suitable for DHTs.

There are several attempts at applying P2P DHT techniques to the retrieval of distributed ontology encoded knowledge information, e.g. in RDF, in semantic overlay networks [11, 15, 23]. In supporting an ontology-driven DHT-based P2P routing mechanism for the KBN, the approach outlined in [15] seems most promising due to its support for peer clustering. Used in this way, peer clustering introduces a hierarchy of peer groups based on policies. Such policies could admit nodes based on semantic closeness, recorded performance, administrative domains or indeed reasoning capabilities. It therefore provides a mechanism for these different routing configuration strategies to co-exist, serving different application domains or user communities in a way that supports incremental deployment and innovation.

Like the preliminary approach taken in this paper, the design presented in [5] uses the semantics of the message and knowledge of the entire network to decide where a subscription should be inserted into the network to minimise the routing table at individual nodes. A slightly different approach in [2] requires the entire network to be searched for a cluster before a subscription is submitted. However, these systems remain CBNs rather than KBNs because the semantics of the message cannot be used in subscriptions and so lacks the expressiveness of the system physically evaluated in this paper. In the presented KBN, it is planned to employ more sophisticated and dynamically reconfigurable clustering schemes, that can be without the need for the complete knowledge of the semantics of all of the network, either before hand or by searching, so improving scalability [29].

7 CONCLUSIONS AND FUTURE WORK

This paper raises some of the scalability issues involved in building a global Knowledge-Based Network. Fundamental to this is the need to support a large array of heterogeneous types in notification messages to accommodate the global variety in

message sources and in the subscriptions to those messages. The performance of a KBN implementation which extends the Siena CBN has been explored. One part of the extension provides ontological concepts as an additional message attribute type, onto which subsumption relationships can be applied. The other part provides for a bag type to be used that allows bag equivalence, sub-bag and super-bag relationships to be used in subscription filters, composed with different bag element comparators. These two extensions augment the expressiveness of CBNs to directly support two major evolutions in the typing of data on the web, the use of ontologies in the Semantics Web and the use of string based tagging and folksonomies in Web 2.0. These evolutions allow the WWW to cope with a dramatic increase in the number of sources of information by providing richer meta-data about content; however the widespread use of rich semantics in meta-data is still not in evidence.

One of the main questions that surround the use of ontologies deep in the network at the routing layer remains the evaluation of the resulting performance overhead. Previous small scale studies in this area [29, 30, 31] show a definite performance penalty, but this may be acceptable when offset against the increased flexibility and expressiveness of the KBN subscription mechanism. Further research is required to evaluate how the performance of "off-the-shelf" ontology tools will affect the scalability of KBNs within larger scales. These results point to the potential importance of semantic clustering for efficient network and performance scalability. It is acknowledged that the experiments in this paper demonstrate only rudimentary semantic clustering. However, the experiments in this paper clearly demonstrate how even inflexible and static clustering can have a substantial positive effect. Ongoing research will focus on how clustering can be performed dynamically as the semantics of the data within the network changes.

Work is also focusing on integrating policy-based cluster management for the KBN [29] to support much more sophisticated cluster schemes, e.g. overlapping clusters and hierarchies of clusters under separate administrative control. Policy-driven clustering enables the size and granularity of peer clusters to reflect different application domains. For example, the clustering policy may be specified in terms of accuracy, latency or reasoning resources as well as the semantic spread of the query-able knowledge-base, or in terms of queries across a peer population and of the querying load across that population. In addition, the effect of semantic interoperability in node matching functions and in inter-cluster communications will be assessed. This requires evaluation of different schemes for injecting newly discovered semantic interoperability mappings into the ontological corpus held by any given cluster, as well as how these mappings are shared between clusters. We expect that any practical system will need to adapt its clustering to reflect the constantly changing profile of semantics being sent and subscribed to via the KBN, thus creating a network environment in which messages are passed from node-to-node, cluster-to-cluster based not on the data's destination but based on the messages semantic data.

Acknowledgement. This work is funded by Science Foundation Ireland under Grant No 05/RFP/CMS014.

REFERENCES

- [1] Web 2.0: Predictions and Pithy Analysis Charles Buchwalter, VP Industry Solutions, Nielsen/NetRatings – Nov 2006
- [2] Anceaume, E., Gradinariu, M., Datta, A. K., Simon, G., Virgillito, A., "A Semantic Overlay for Self- Peer-to-Peer Publish/Subscribe", Int'l Conf. on Distributed Computing Systems (ICDCS'06), 4-7 July 2006, Lisboa, Portugal.
- [3] "Pew Internet Project Data memo", Mary Madden, Pew Internet and American Life Project, Nov 2006
- [4] "The Economics of Podcasting", Nielson Media Research, July 2006
- [5] Chand, R., Felber, P., "Semantic Peer-to-Peer Overlays for Publish/Subscribe Networks", EuroPar 2005, European Conference on Parallel Processing, Lisboa, Portugal, 2005.
- [6] Chand, R., Felber, P., Garofalakis, M., "Tree-Pattern Similarity Estimation for Scalable content-based Routing", ICDE 2007, Int'l Conf. on Data Engineering, Istanbul, Turkey, 16-20 April, 2007.
- [7] Apple - iTunes - iTunes Store - Podcasts - Technical Specification. Retrieved March 18th, 2007 from: www.apple.com/itunes/podcasts/techspecs.html
- [8] Swoogle's Statistics of the Semantic Web Retrieved March 18th, 2007 from: http://swoogle.umbc.edu/index.php?option=com_swoogle_stats&Itemid=8
- [9] "RSS - Crossing Into the Mainstream", Joshua Grossnickle, Yahoo! White Paper, Oct 2005
- [10] Peterson, L., et al, "Experiences Building PlanetLab", Symposium on Operating System Design and Implementation (OSDI '06) - Nov 2006
- [11] Cai, M., Frank, M., "RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network", WWW conference, May 2004, New York, USA.
- [12] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2001). Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems, 19(3).
- [13] Weisstein, E. W. (2002). Multiset. MathWorld – A Wolfram Web Resource. Retrieved July 19, 2006, from <http://mathworld.wolfram.com/Multiset.html>.
- [14] Jiang J, Conrath D., "Semantic Similarity based on corpus statistics and lexical taxonomy", Intl Conference on Research in Computational Linguistics, 1997.
- [15] Loser, A., Naumann, F., Siberski, W., Nejdil, W., Thaden, U., "Semantic overlay clusters within super-peer networks", Int'l Workshop on Databases, Information Systems and Peer-to-Peer Computing in Conjunction with the VLDB 2003
- [16] Lynch, D., Keeney, J., Lewis, D., O'Sullivan, D., "A Proactive Approach to Semantically Oriented Service Discovery". Innovations in Web Infrastructure (IWI 2006). at World-Wide Web Conf., Edinburgh, Scotland. May 2006.
- [17] Li, H., Jiang, G., "Semantic Message Oriented Middleware for Publish/Subscribe Networks", in proc of SPIE, Volume 5403, pp 124-133, 2004
- [18] Muthusamy, V., Jacobsen, H.A., "Small-scale peer-to-peer publish/subscribe" in proc Workshop on Peer-to-Peer Knowledge Management, San Diego, USA, July 2005
- [19] Rada R., Mili H., Bicknell E., Blettner M., "Development and application of a metric on semantic nets", IEEE Transactions on Systems, Man, and Cybernetics 19, 1989.
- [20] Rutherford, M. J. (2004). "Siena Simplification Library Documentation 1.1.4." University of Colorado – Web Resource. Retrieved August 13, 2006, from <http://serl.cs.colorado.edu/carzanig/siena/forwarding/ssimp/namespacesiena.html>
- [21] Segall, B. et al, "Content-Based Routing in Elvin4", In proc AUUG2K, Canberra 2000.
- [22] Pietzuch, P., Bacon, J., "Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware". Distributed Event-Based Systems (DEBS'03). At the ACM SIGMOD/PODS Conference, San Diego, California, June 2003
- [23] Tempich, C., Staab, S., Wranik, A., "REMINDIN': semantic query routing in peer-to-peer networks based on social metaphors" WWW 2004, New York, USA, 2004.
- [24] Terpstra, W.W., Behnel, S., Fiege, L., Zeidler, A., Buchmann, A.P., "A peer-to-peer approach to content-based publish/subscribe", in proc of DEBS 2003, ACM Press 2003
- [25] Keeney, J., Lewis, D., O'Sullivan, D., "Benchmarking Knowledge-based Context Delivery Systems", in proc of ICAS 06, Silicon Valley, USA, July 19-21, 2006.
- [26] Carzaniga, A., Wolf, A. L., "A Benchmark Suite for Distributed Publish/Subscribe Systems", Technical Report CU-CS-927-02, Dept. of Computer Science, University of Colorado. Apr 2002, <http://serl.cs.colorado.edu/~carzanig/papers/>
- [27] Crowcroft, J., Bacon, J., Pietzuch, P., Coulouris, G., Naguib, H., "Channel Islands in a Reflective Ocean: Large-Scale Event Distribution in Heterogeneous Networks", IEEE Communications, Vol 40 No. 9, Sept 2002.
- [28] Petrovic, M., Burceaa, I., Jacobsen, H.A. "S-ToPSS – a semantic publish/subscribe system" in proc VLDB, Berlin, Germany, September 2003
- [29] Lewis, D., Keeney, J., O'Sullivan, D., Guo, S., "Towards a Managed Extensible Control Plane for Knowledge-Based Networking", Distributed Systems: Operations and Management Large Scale Management, (DSOM 2006), at Manweek 2006, Dublin, Ireland, 23-25 October 2006
- [30] Keeney, J., Lewis, D., O'Sullivan, D., Roelens, A., Boran, A., Richardson, R., "Runtime Semantic Interoperability for Gathering Ontology-based Network Context", Network Operations and Management Symposium (NOMS 2006), Vancouver, Canada. 3-7 April 2006.
- [31] Keeney, J., Lewis, D., O'Sullivan, D., "Ontological Semantics for Distributing Contextual Knowledge in Highly Distributed Autonomic Systems", Journal of Network and System Management, Vol 15, March 2007
- [32] Muhl, G., Fiege, L., Gartner, F., Buchman, A., "Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems", Int'l Symp. On Modelling, Analysis and Simulation of Computer Telecommunications Systems (MASCOT'02), 2002
- [33] Chand, R., Felber, P., "XNet: a Reliable Content Based Publish Subscribe System". SRDS 2004, Symp. on Reliable Distributed Systems, Florianopolis, Brazil, October 2004.
- [34] Roblek, D., "Decentralized Discovery and Execution for Composite Semantic Web Services", M.Sc. Thesis, Computer Science, Trinity College Dublin, Ireland, December 2006.
- [35] Cilia, M., Bornhövd, C., Buchmann, A. P., "CREAM: An Infrastructure for Distributed, Heterogeneous Event-Based Applications". CoopIS 2003, Catania, Sicily, Italy, Nov2003
- [36] Carzaniga, A., Wolf, A. L., "Forwarding in a Content-Based Network" SIGCOMM'03, Kaellsruhe, Germany. August 2003

Appendix B

confOf.txt

Page 1 of 2

1: Suggested that MEDOID: Camera_Ready_event joins cluster: http://confOf#Administrative_event
2: Suggested that MEDOID: Social_event joins cluster: http://confOf#Social_event
3: Suggested that MEDOID: South_America joins cluster: http://confOf#Country
4: Suggested that MEDOID: Assistant joins cluster: http://confOf#Person
5: Suggested that MEDOID: Company joins cluster: http://confOf#Organization
6: Suggested that MEDOID: Science_Worker joins cluster: http://confOf#Person
7: Suggested that MEDOID: Scholar joins cluster: http://confOf#Person
8: Suggested that MEDOID: Administrator joins cluster: http://confOf#Organization
9: Suggested that MEDOID: Event joins cluster: http://confOf#Event
10: Suggested that MEDOID: Topic joins cluster: http://confOf#Topic
11: Suggested that MEDOID: Asia joins cluster: http://confOf#Country
12: Suggested that MEDOID: Tutorial joins cluster: http://confOf#Working_event
13: Suggested that MEDOID: Trip joins cluster: http://confOf#Social_event
14: Suggested that MEDOID: Submission_event joins cluster: http://confOf#Administrative_event
15: Suggested that MEDOID: Organization joins cluster: http://confOf#Organization
16: Suggested that MEDOID: Participant joins cluster: http://confOf#Participant
17: Suggested that MEDOID: Student joins cluster: http://confOf#Participant
18: Suggested that MEDOID: Member joins cluster: http://confOf#Participant
19: Suggested that MEDOID: North_America joins cluster: http://confOf#Country
20: Suggested that MEDOID: Short_paper joins cluster: http://confOf#Contribution
21: Suggested that MEDOID: Author joins cluster: http://confOf#Person
22: Suggested that MEDOID: Workshop joins cluster: http://confOf#Working_event
23: Suggested that MEDOID: Volunteer joins cluster: http://confOf#Organization
24: Suggested that MEDOID: Africa joins cluster: http://confOf#Country
25: Suggested that MEDOID: Reviewing_event joins cluster: http://confOf#Administrative_event
26: Suggested that MEDOID: Working_event joins cluster: http://confOf#Event
27: Suggested that MEDOID: Contribution joins cluster: http://confOf#Contribution
28: Suggested that MEDOID: Administrative_event joins cluster: http://confOf#Event
29: Suggested that MEDOID: Paper joins cluster: http://confOf#Contribution
30: Suggested that MEDOID: Conference joins cluster: http://confOf#Working_event
31: Suggested that MEDOID: Person joins cluster: http://confOf#Person
32: Suggested that MEDOID: Registration_of_participants_event joins cluster: http://confOf#Administrative_event
33: Suggested that MEDOID: Country joins cluster: http://confOf#Country
34: Suggested that MEDOID: Banquet joins cluster: http://confOf#Social_event
35: Suggested that MEDOID: City joins cluster: http://confOf#City
36: Suggested that MEDOID: University joins cluster: http://confOf#Organization
37: Suggested that MEDOID: Reviewing_results_event joins cluster: http://confOf#Administrative_event
38: Suggested that MEDOID: Australisa joins cluster: http://confOf#Country
39: Suggested that MEDOID: Regular joins cluster: http://confOf#Participant
40: Suggested that MEDOID: Reception joins cluster: http://confOf#Social_event
41: Suggested that MEDOID: Poster joins cluster: http://confOf#Contribution
42: Suggested that MEDOID: Chair_PC joins cluster: http://confOf#Organization
43: Suggested that MEDOID: Europe joins cluster: http://confOf#Country
44: Suggested that MEDOID: Member_PC joins cluster: http://confOf#Organization
45: Suggested that MEDOID: Autonomic_Actions joins cluster: http://confOf#Topic
46: Suggested that MEDOID: Carlow joins cluster: http://confOf#City
47: Suggested that MEDOID: Ireland joins cluster: http://confOf#Country
48: Suggested that MEDOID: Middleware joins cluster: http://confOf#Topic
49: Suggested that MEDOID: Academic_Andrew joins cluster: http://confOf#Person
50: Suggested that MEDOID: Keeney_John joins cluster: http://confOf#Person

confOf.txt

Page 2 of 2

51: Suggested that MEDOID: Dublin joins cluster: http://confOf#City
52: Suggested that MEDOID: Dynaism joins cluster: http://confOf#Topic
53: Suggested that MEDOID: Fomal_Methods joins cluster: http://confOf#Topic
54: Suggested that MEDOID: Galway joins cluster: http://confOf#City
55: Suggested that MEDOID: Short_paper_19 joins cluster: http://confOf#Contribution
56: Suggested that MEDOID: Jones_Dominic joins cluster: http://confOf#Person
57: Suggested that MEDOID: Poster_1 joins cluster: http://confOf#Contribution
58: Suggested that MEDOID: Policy-based_Network_Managment joins cluster: http://confOf#Topic
59: Suggested that MEDOID: Rome joins cluster: http://confOf#City
60: Suggested that MEDOID: Trinity_College_Dublin joins cluster: http://confOf#Organization
61: Suggested that MEDOID: Paper_17 joins cluster: http://confOf#Contribution
62: Suggested that MEDOID: Fortaleza joins cluster: http://confOf#City
63: Suggested that MEDOID: Lewis_Dave joins cluster: http://confOf#Person

1: Suggested that MEDOID: Invited_Talk_Abstract joins cluster: <http://ekaw#Abstract>
2: Suggested that MEDOID: Assigned_Paper joins cluster: http://ekaw#Assigned_Paper
3: Suggested that MEDOID: Multi-author_Volume joins cluster: http://ekaw#Multi-author_Volume
4: Suggested that MEDOID: Organisation joins cluster: <http://ekaw#Organisation>
5: Suggested that MEDOID: Location joins cluster: <http://ekaw#Location>
6: Suggested that MEDOID: Invited_Speaker joins cluster: <http://ekaw#Presenter>
7: Suggested that MEDOID: Submitted_Paper joins cluster: http://ekaw#Submitted_Paper
8: Suggested that MEDOID: Demo_Chair joins cluster: http://ekaw#Conference_Participant
9: Suggested that MEDOID: Poster_Session joins cluster: <http://ekaw#Session>
10: Suggested that MEDOID: Early-Registered_Participant joins cluster: http://ekaw#Conference_Participant
11: Suggested that MEDOID: PC_Chair joins cluster: http://ekaw#PC_Member
12: Suggested that MEDOID: OC_Chair joins cluster: http://ekaw#OC_Member
13: Suggested that MEDOID: Tutorial_Chair joins cluster: http://ekaw#PC_Member
14: Suggested that MEDOID: Track joins cluster: http://ekaw#Scientific_Event
15: Suggested that MEDOID: Evaluated_Paper joins cluster: http://ekaw#Assigned_Paper
16: Suggested that MEDOID: Abstract joins cluster: <http://ekaw#Abstract>
17: Suggested that MEDOID: Programme_Brochure joins cluster: <http://ekaw#Document>
18: Suggested that MEDOID: Tutorial_Abstract joins cluster: <http://ekaw#Abstract>
19: Suggested that MEDOID: Organising_Agency joins cluster: <http://ekaw#Organisation>
20: Suggested that MEDOID: Agency_Staff_Member joins cluster: <http://ekaw#Person>
21: Suggested that MEDOID: PC_Member joins cluster: http://ekaw#PC_Member
22: Suggested that MEDOID: Research_Topic joins cluster: http://ekaw#Research_Topic
23: Suggested that MEDOID: Proceedings joins cluster: http://ekaw#Multi-author_Volume
24: Suggested that MEDOID: Workshop_Session joins cluster: <http://ekaw#Session>
25: Suggested that MEDOID: Regular_Session joins cluster: <http://ekaw#Session>
26: Suggested that MEDOID: Web_Site joins cluster: <http://ekaw#Document>
27: Suggested that MEDOID: Session joins cluster: <http://ekaw#Session>
28: Suggested that MEDOID: SC_Member joins cluster: http://ekaw#PC_Member
29: Suggested that MEDOID: Workshop_Chair joins cluster: http://ekaw#PC_Member
30: Suggested that MEDOID: Negative_Review joins cluster: <http://ekaw#Review>
31: Suggested that MEDOID: Workshop joins cluster: http://ekaw#Scientific_Event
32: Suggested that MEDOID: Conference_Paper joins cluster: <http://ekaw#Paper>
33: Suggested that MEDOID: Research_Institute joins cluster: http://ekaw#Academic_Institution
34: Suggested that MEDOID: Possible_Reviewer joins cluster: http://ekaw#Possible_Reviewer
35: Suggested that MEDOID: Scientific_Event joins cluster: http://ekaw#Scientific_Event
36: Suggested that MEDOID: Document joins cluster: <http://ekaw#Document>
37: Suggested that MEDOID: Workshop_Paper joins cluster: <http://ekaw#Paper>
38: Suggested that MEDOID: Person joins cluster: <http://ekaw#Person>
39: Suggested that MEDOID: Individual_Presentation joins cluster: http://ekaw#Scientific_Event
40: Suggested that MEDOID: Rejected_Paper joins cluster: http://ekaw#Evaluated_Paper
41: Suggested that MEDOID: Session_Chair joins cluster: http://ekaw#PC_Member
42: Suggested that MEDOID: Proceedings_Publisher joins cluster: <http://ekaw#Organisation>
43: Suggested that MEDOID: Academic_Institution joins cluster: <http://ekaw#Organisation>
44: Suggested that MEDOID: Flyer joins cluster: <http://ekaw#Document>
45: Suggested that MEDOID: Review joins cluster: <http://ekaw#Document>
46: Suggested that MEDOID: Industrial_Session joins cluster: http://ekaw#Conference_Session
47: Suggested that MEDOID: Demo_Session joins cluster: <http://ekaw#Session>
48: Suggested that MEDOID: Camera_Ready_Paper joins cluster: <http://ekaw#Paper>
49: Suggested that MEDOID: Accepted_Paper joins cluster: http://ekaw#Evaluated_Paper
50: Suggested that MEDOID: University joins cluster: http://ekaw#Academic_Institution

51: Suggested that MEDOID: Conference joins cluster: http://ekaw#Scientific_Event
52: Suggested that MEDOID: Positive_Review joins cluster: <http://ekaw#Review>
53: Suggested that MEDOID: Presenter joins cluster: <http://ekaw#Presenter>
54: Suggested that MEDOID: Social_Event joins cluster: http://ekaw#Social_Event
55: Suggested that MEDOID: Demo_Paper joins cluster: <http://ekaw#Paper>
56: Suggested that MEDOID: Conference_Proceedings joins cluster: <http://ekaw#Proceedings>
57: Suggested that MEDOID: Contributed_Talk joins cluster: http://ekaw#Individual_Presentation
58: Suggested that MEDOID: OC_Member joins cluster: http://ekaw#OC_Member
59: Suggested that MEDOID: Paper_Author joins cluster: <http://ekaw#Person>
60: Suggested that MEDOID: Student joins cluster: <http://ekaw#Person>
61: Suggested that MEDOID: Tutorial joins cluster: http://ekaw#Individual_Presentation
62: Suggested that MEDOID: Poster_Paper joins cluster: <http://ekaw#Paper>
63: Suggested that MEDOID: Conference_Trip joins cluster: http://ekaw#Social_Event
64: Suggested that MEDOID: Regular_Paper joins cluster: <http://ekaw#Paper>
65: Suggested that MEDOID: Late-Registered_Participant joins cluster: http://ekaw#Conference_Participant
66: Suggested that MEDOID: Invited_Talk joins cluster: http://ekaw#Individual_Presentation
67: Suggested that MEDOID: Paper joins cluster: <http://ekaw#Document>
68: Suggested that MEDOID: Conference_Participant joins cluster: <http://ekaw#Person>
69: Suggested that MEDOID: Industrial_Paper joins cluster: <http://ekaw#Paper>
70: Suggested that MEDOID: Conference_Session joins cluster: <http://ekaw#Session>
71: Suggested that MEDOID: Neutral_Review joins cluster: <http://ekaw#Review>
72: Suggested that MEDOID: Conference_Banquet joins cluster: http://ekaw#Social_Event
73: Suggested that MEDOID: Event joins cluster: <http://ekaw#Event>

1: Suggested that MEDOID: Event joins cluster: <http://annotation.semanticweb.org/2004/iswc#Event>
2: Suggested that MEDOID: Formal_Language joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
3: Suggested that MEDOID: University joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
4: Suggested that MEDOID: Faculty_Member joins cluster: http://annotation.semanticweb.org/2004/iswc#Faculty_Member
5: Suggested that MEDOID: Application_Domain joins cluster: http://annotation.semanticweb.org/2004/iswc#Application_Domain
6: Suggested that MEDOID: Workshop joins cluster: <http://annotation.semanticweb.org/2004/iswc#Event>
7: Suggested that MEDOID: Employee joins cluster: <http://annotation.semanticweb.org/2004/iswc#Person>
8: Suggested that MEDOID: Associate_Professor joins cluster: http://annotation.semanticweb.org/2004/iswc#Faculty_Member
9: Suggested that MEDOID: Institute joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
10: Suggested that MEDOID: Lecturer joins cluster: http://annotation.semanticweb.org/2004/iswc#Faculty_Member
11: Suggested that MEDOID: Project joins cluster: <http://annotation.semanticweb.org/2004/iswc#Project>
12: Suggested that MEDOID: Application joins cluster: <http://annotation.semanticweb.org/2004/iswc#Application>
13: Suggested that MEDOID: Association joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
14: Suggested that MEDOID: Tutorial joins cluster: <http://annotation.semanticweb.org/2004/iswc#Event>
15: Suggested that MEDOID: Topic joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
16: Suggested that MEDOID: Organization joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
17: Suggested that MEDOID: Research_Funding_Institution joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
18: Suggested that MEDOID: Method joins cluster: <http://annotation.semanticweb.org/2004/iswc#Method>
19: Suggested that MEDOID: Department joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
20: Suggested that MEDOID: Tool joins cluster: <http://annotation.semanticweb.org/2004/iswc#Tool>
21: Suggested that MEDOID: Book joins cluster: <http://annotation.semanticweb.org/2004/iswc#Publication>
22: Suggested that MEDOID: Algorithm joins cluster: <http://annotation.semanticweb.org/2004/iswc#Algorithm>
23: Suggested that MEDOID: Report joins cluster: <http://annotation.semanticweb.org/2004/iswc#Publication>
24: Suggested that MEDOID: Person joins cluster: <http://annotation.semanticweb.org/2004/iswc#Person>
25: Suggested that MEDOID: Student joins cluster: <http://annotation.semanticweb.org/2004/iswc#Person>
26: Suggested that MEDOID: Full_Professor joins cluster: http://annotation.semanticweb.org/2004/iswc#Faculty_Member
27: Suggested that MEDOID: InProceedings joins cluster: <http://annotation.semanticweb.org/2004/iswc#Publication>
28: Suggested that MEDOID: PhDstudent joins cluster: <http://annotation.semanticweb.org/2004/iswc#Student>
29: Suggested that MEDOID: Conference joins cluster: <http://annotation.semanticweb.org/2004/iswc#Event>
30: Suggested that MEDOID: Researcher joins cluster: <http://annotation.semanticweb.org/2004/iswc#Person>
31: Suggested that MEDOID: Proceedings joins cluster: <http://annotation.semanticweb.org/2004/iswc#Publication>
32: Suggested that MEDOID: Publication joins cluster: <http://annotation.semanticweb.org/2004/iswc#Publication>
33: Suggested that MEDOID: Enterprise joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
34: Suggested that MEDOID: RDF joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
35: Suggested that MEDOID: Web_Services joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
36: Suggested that MEDOID: Java joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
37: Suggested that MEDOID: Machine_Learning joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
38: Suggested that MEDOID: C joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
39: Suggested that MEDOID: Knowledge_Discovery joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
40: Suggested that MEDOID: SQL joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
41: Suggested that MEDOID: e-Business joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
42: Suggested that MEDOID: Ontology_Learning joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
43: Suggested that MEDOID: Semantic_Web_Infrastructure joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
44: Suggested that MEDOID: Agents joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
45: Suggested that MEDOID: OWL joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
46: Suggested that MEDOID: Knowledge_Systems joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
47: Suggested that MEDOID: Semantic_Web_Languages joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
48: Suggested that MEDOID: Business_Engineering joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
49: Suggested that MEDOID: KAMN joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
50: Suggested that MEDOID: Ontology-based_Knowledge_Management_Systems joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>

51: Suggested that MEDOID: RDF5 joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
52: Suggested that MEDOID: Logic joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
53: Suggested that MEDOID: University_of_Karlsruhe joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
54: Suggested that MEDOID: Office_Information_Systems joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
55: Suggested that MEDOID: Information_Retrieval joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
56: Suggested that MEDOID: Network_Infrastructure joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
57: Suggested that MEDOID: DAML_OIL joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
58: Suggested that MEDOID: Semantic_Web joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
59: Suggested that MEDOID: TowardsSemanticWebMining joins cluster: <http://annotation.semanticweb.org/2004/iswc#Publication>
60: Suggested that MEDOID: Modeling joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
61: Suggested that MEDOID: Artificial_Intelligence joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
62: Suggested that MEDOID: Semantic_Annotation joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
63: Suggested that MEDOID: Ontology_Engineering joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
64: Suggested that MEDOID: Knowledge_Management joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
65: Suggested that MEDOID: Query_Languages joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
66: Suggested that MEDOID: Information_Extraction joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
67: Suggested that MEDOID: XML joins cluster: http://annotation.semanticweb.org/2004/iswc#Formal_Language
68: Suggested that MEDOID: Agent_Systems joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
69: Suggested that MEDOID: Text_Mining joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
70: Suggested that MEDOID: AIFB joins cluster: <http://annotation.semanticweb.org/2004/iswc#Organization>
71: Suggested that MEDOID: Knowledge_Representation_Languages joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
72: Suggested that MEDOID: Data_Mining joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
73: Suggested that MEDOID: Knowledge_Representation_And_Reasoning joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
74: Suggested that MEDOID: World_Wide_Web joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
75: Suggested that MEDOID: Knowledge_Reasoning joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
76: Suggested that MEDOID: Human_Computer_Interaction joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
77: Suggested that MEDOID: Matching joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
78: Suggested that MEDOID: Databases joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
79: Suggested that MEDOID: ISWC_2002 joins cluster: <http://annotation.semanticweb.org/2004/iswc#Event>
80: Suggested that MEDOID: Information_Systems joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
81: Suggested that MEDOID: Knowledge_Management_Methodology joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
82: Suggested that MEDOID: Development_of_Knowledge_Management_Systems joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>
83: Suggested that MEDOID: Knowledge_Portals joins cluster: <http://annotation.semanticweb.org/2004/iswc#Topic>

- 1: Suggested that MEDOID: InProceedings joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 2: Suggested that MEDOID: TechnicalReport joins cluster: <http://swrc.ontaware.org/ontology#Report>
- 3: Suggested that MEDOID: Student joins cluster: <http://swrc.ontaware.org/ontology#Student>
- 4: Suggested that MEDOID: Lecture joins cluster: <http://swrc.ontaware.org/ontology#Event>
- 5: Suggested that MEDOID: FullProfessor joins cluster: <http://swrc.ontaware.org/ontology#FacultyMember>
- 6: Suggested that MEDOID: ResearchProject joins cluster: <http://swrc.ontaware.org/ontology#Project>
- 7: Suggested that MEDOID: Conference joins cluster: <http://swrc.ontaware.org/ontology#Event>
- 8: Suggested that MEDOID: Graduate joins cluster: <http://swrc.ontaware.org/ontology#Student>
- 9: Suggested that MEDOID: Article joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 10: Suggested that MEDOID: PhDThesis joins cluster: <http://swrc.ontaware.org/ontology#Thesis>
- 11: Suggested that MEDOID: Manual joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 12: Suggested that MEDOID: Meeting joins cluster: <http://swrc.ontaware.org/ontology#Meeting>
- 13: Suggested that MEDOID: Topic joins cluster: <http://swrc.ontaware.org/ontology#Topic>
- 14: Suggested that MEDOID: Institute joins cluster: <http://swrc.ontaware.org/ontology#Organization>
- 15: Suggested that MEDOID: ResearchGroup joins cluster: <http://swrc.ontaware.org/ontology#Organization>
- 16: Suggested that MEDOID: Association joins cluster: <http://swrc.ontaware.org/ontology#Organization>
- 17: Suggested that MEDOID: Lecturer joins cluster: <http://swrc.ontaware.org/ontology#AcademicStaff>
- 18: Suggested that MEDOID: Product joins cluster: <http://swrc.ontaware.org/ontology#Product>
- 19: Suggested that MEDOID: ProjectMeeting joins cluster: <http://swrc.ontaware.org/ontology#Meeting>
- 20: Suggested that MEDOID: Department joins cluster: <http://swrc.ontaware.org/ontology#Organization>
- 21: Suggested that MEDOID: AssociateProfessor joins cluster: <http://swrc.ontaware.org/ontology#FacultyMember>
- 22: Suggested that MEDOID: Person joins cluster: <http://swrc.ontaware.org/ontology#Person>
- 23: Suggested that MEDOID: FacultyMember joins cluster: <http://swrc.ontaware.org/ontology#FacultyMember>
- 24: Suggested that MEDOID: University joins cluster: <http://swrc.ontaware.org/ontology#Organization>
- 25: Suggested that MEDOID: Exhibition joins cluster: <http://swrc.ontaware.org/ontology#Event>
- 26: Suggested that MEDOID: Employee joins cluster: <http://swrc.ontaware.org/ontology#Person>
- 27: Suggested that MEDOID: MasterThesis joins cluster: <http://swrc.ontaware.org/ontology#Thesis>
- 28: Suggested that MEDOID: SoftwareComponent joins cluster: <http://swrc.ontaware.org/ontology#Product>
- 29: Suggested that MEDOID: Book joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 30: Suggested that MEDOID: Report joins cluster: <http://swrc.ontaware.org/ontology#Report>
- 31: Suggested that MEDOID: PhDStudent joins cluster: <http://swrc.ontaware.org/ontology#Graduate>
- 32: Suggested that MEDOID: TechnicalStaff joins cluster: <http://swrc.ontaware.org/ontology#Employee>
- 33: Suggested that MEDOID: Manager joins cluster: <http://swrc.ontaware.org/ontology#Employee>
- 34: Suggested that MEDOID: Undergraduate joins cluster: <http://swrc.ontaware.org/ontology#Student>
- 35: Suggested that MEDOID: Project joins cluster: <http://swrc.ontaware.org/ontology#Project>
- 36: Suggested that MEDOID: Misc joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 37: Suggested that MEDOID: Event joins cluster: <http://swrc.ontaware.org/ontology#Event>
- 38: Suggested that MEDOID: InBook joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 39: Suggested that MEDOID: ResearchTopic joins cluster: <http://swrc.ontaware.org/ontology#Topic>
- 40: Suggested that MEDOID: Publication joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 41: Suggested that MEDOID: Proceedings joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 42: Suggested that MEDOID: ProjectReport joins cluster: <http://swrc.ontaware.org/ontology#Report>
- 43: Suggested that MEDOID: Unpublished joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 44: Suggested that MEDOID: Workshop joins cluster: <http://swrc.ontaware.org/ontology#Event>
- 45: Suggested that MEDOID: Booklet joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 46: Suggested that MEDOID: AssistantProfessor joins cluster: <http://swrc.ontaware.org/ontology#FacultyMember>
- 47: Suggested that MEDOID: Thesis joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 48: Suggested that MEDOID: InCollection joins cluster: <http://swrc.ontaware.org/ontology#Publication>
- 49: Suggested that MEDOID: AcademicStaff joins cluster: <http://swrc.ontaware.org/ontology#Employee>
- 50: Suggested that MEDOID: Organization joins cluster: <http://swrc.ontaware.org/ontology#Organization>

- 51: Suggested that MEDOID: AdministrativeStaff joins cluster: <http://swrc.ontaware.org/ontology#Employee>
- 52: Suggested that MEDOID: Enterprise joins cluster: <http://swrc.ontaware.org/ontology#Organization>
- 53: Suggested that MEDOID: SoftwareProject joins cluster: <http://swrc.ontaware.org/ontology#DevelopmentProject>
- 54: Suggested that MEDOID: DevelopmentProject joins cluster: <http://swrc.ontaware.org/ontology#Project>

1: Suggested that MEDOID: IPPacket joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkLayerPacket>
2: Suggested that MEDOID: ProgramFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_File
3: Suggested that MEDOID: HostOnInternet joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost>
4: Suggested that MEDOID: Scavenging joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PassiveAttack>
5: Suggested that MEDOID: MACAlgorithm joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MACAlgorithm>
6: Suggested that MEDOID: Recovery joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MechanismType>
7: Suggested that MEDOID: Credential joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Credential>
8: Suggested that MEDOID: Heap joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DataOnVolatileMedia>
9: Suggested that MEDOID: FormatStringAttack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MalformedInput>
10: Suggested that MEDOID: HostOnWireNetwork joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost>
11: Suggested that MEDOID: LibertyFramework joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SingleSignOnSystem>
12: Suggested that MEDOID: CertificateData joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#CertificateData>
13: Suggested that MEDOID: WiredNetwork joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Network>
14: Suggested that MEDOID: Checksum joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Checksum>
15: Suggested that MEDOID: Model joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Model>
16: Suggested that MEDOID: Availability joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasGoal>
17: Suggested that MEDOID: MD4 joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#CryptographicHashFunction>
18: Suggested that MEDOID: Hardware joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Hardware>
19: Suggested that MEDOID: Damage joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PhysicalThreat>
20: Suggested that MEDOID: DiscreteLogarithm joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DiscreteLogarithm>
21: Suggested that MEDOID: NetworkedHost joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost>
22: Suggested that MEDOID: BiometricAuthenticationSystem joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#BiometricAuthenticationSystem>
23: Suggested that MEDOID: Iris joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential>
24: Suggested that MEDOID: DataInTransit joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_DataInTransit
25: Suggested that MEDOID: Trust joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasGoal>
26: Suggested that MEDOID: Least-Privilege joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#AccessControlModel>
27: Suggested that MEDOID: Harddisk joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Hardware>
28: Suggested that MEDOID: Rootkit joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MaliciousCode>
29: Suggested that MEDOID: Technology joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Technology>
30: Suggested that MEDOID: MandatoryAccessControl joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#AccessControlModel>
31: Suggested that MEDOID: UseOfVulnerableProgrammingLanguage joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#UseOfVulnerableProgrammingLanguage>
32: Suggested that MEDOID: RIPE-MAC3 joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#RIPE-MAC>
33: Suggested that MEDOID: Definition joins cluster: <http://www.ida.liu.se/~almhe/SecurityLiterature.owl#Definition>
34: Suggested that MEDOID: Skipjack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SymmetricAlgorithm>
35: Suggested that MEDOID: ConfigurationFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_File
36: Suggested that MEDOID: MalformedInput joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MalformedInput>
37: Suggested that MEDOID: Human joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Human>
38: Suggested that MEDOID: MessageDigestData joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Credential>
39: Suggested that MEDOID: Cryptography joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Cryptography>
40: Suggested that MEDOID: MessageDigest joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MessageDigest>
41: Suggested that MEDOID: AnomalyDetectionSystem joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#ByDetectionMethod>
42: Suggested that MEDOID: ClientHost joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost>
43: Suggested that MEDOID: Receiver joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Human>
44: Suggested that MEDOID: KeyExchange joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Cryptography>
45: Suggested that MEDOID: LoginName joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Credential
46: Suggested that MEDOID: Credential joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Credential
47: Suggested that MEDOID: DataOnNon-VolatileMedia joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_DataOnNon-VolatileMedia
48: Suggested that MEDOID: OnionRouter joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SecureNetworkCommunication>
49: Suggested that MEDOID: Anonymity joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#ConfidentialitySimilaris>

50: Suggested that MEDOID: StackOverflow joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#BufferOverflow>
51: Suggested that MEDOID: TrojanHorse joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MaliciousCode>
52: Suggested that MEDOID: WirelessAccessPoint joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Router>
53: Suggested that MEDOID: Worm joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Worm>
54: Suggested that MEDOID: _2TDES joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#TripleDES>
55: Suggested that MEDOID: Human joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Human>
56: Suggested that MEDOID: NAT joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SecureNetworkCommunication>
57: Suggested that MEDOID: Detection joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MechanismType>
58: Suggested that MEDOID: Network joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Technology>
59: Suggested that MEDOID: Data joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Technology>
60: Suggested that MEDOID: ClientHost joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Host
61: Suggested that MEDOID: PassiveAttack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PassiveAttack>
62: Suggested that MEDOID: RaceCondition joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#RaceCondition>
63: Suggested that MEDOID: BiometricCredential joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Credential
64: Suggested that MEDOID: MPassport joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SingleSignOnSystem>
65: Suggested that MEDOID: Monitoring joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Monitoring>
66: Suggested that MEDOID: Factorisation joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#CryptographyModel>
67: Suggested that MEDOID: Spyware joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Spyware>
68: Suggested that MEDOID: SigningWithCertificate joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SigningWithCertificate>
69: Suggested that MEDOID: Integrity joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasGoal>
70: Suggested that MEDOID: One-TimePassword joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Password>
71: Suggested that MEDOID: Product joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Product>
72: Suggested that MEDOID: ReferenceMonitor joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#AccessControlMechanism>
73: Suggested that MEDOID: StreamCipher joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#StreamCipher>
74: Suggested that MEDOID: MD2 joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#CryptographicHashFunction>
75: Suggested that MEDOID: Router joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost>
76: Suggested that MEDOID: UntrustedNetwork joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Network>
77: Suggested that MEDOID: EncryptionKey joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Credential>
78: Suggested that MEDOID: DigitalSigning joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Checksum>
79: Suggested that MEDOID: DiscretionaryAccessControl joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#AccessControlModel>
80: Suggested that MEDOID: SecurityHardware joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Hardware>
81: Suggested that MEDOID: DNSSEC joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Standard>
82: Suggested that MEDOID: InternetInfrastructureDenialOfService joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#InternetInfrastructureDenialOfService>
83: Suggested that MEDOID: X.509CertificateData joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#CertificateData>
84: Suggested that MEDOID: RIPE-MAC1 joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#RIPE-MAC>
85: Suggested that MEDOID: SSL joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Standard>
86: Suggested that MEDOID: Smurf joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DenialOfService>
87: Suggested that MEDOID: ContinuousIntrusionDetectionSystem joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#ByUsageFrequency>
88: Suggested that MEDOID: Stack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DataOnVolatileMedia>
89: Suggested that MEDOID: StationaryData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_StationaryData
90: Suggested that MEDOID: CPU joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Hardware>
91: Suggested that MEDOID: Blowfish joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Standard>
92: Suggested that MEDOID: HasGoal joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasGoal>
93: Suggested that MEDOID: StackAttack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DenialOfService>
94: Suggested that MEDOID: HMAC joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MACAlgorithm>
95: Suggested that MEDOID: Stack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DataOnVolatileMedia>
96: Suggested that MEDOID: Technology joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Technology
97: Suggested that MEDOID: Receiver joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Human>
98: Suggested that MEDOID: PHPInjection joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#CodeInjection>

99: Suggested that MEDOID: Sanitizer joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Sanitizer
 100: Suggested that MEDOID: DefenseStrategy joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DefenseStrategy
 101: Suggested that MEDOID: Vulnerability joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Vulnerability
 102: Suggested that MEDOID: ConfidentialitySimilarities joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasGoal
 103: Suggested that MEDOID: Keynote joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TrustManagement
 104: Suggested that MEDOID: LoginSystem joins cluster: http://www.ida.liu.se/~almhe/Security.owl#LoginSystem
 105: Suggested that MEDOID: SecurityHardware joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SecurityHardware
 106: Suggested that MEDOID: DataOnNon-VolatileMedia joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DataOnNon-VolatileMedia
 107: Suggested that MEDOID: Authorisation joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasGoal
 108: Suggested that MEDOID: ActiveIntrusionDetectionSystem joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ByBehaviourOnDetection
 109: Suggested that MEDOID: Role-BasedAccessControl joins cluster: http://www.ida.liu.se/~almhe/Security.owl#AccessControlModel
 110: Suggested that MEDOID: BastionHost joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Host
 111: Suggested that MEDOID: CPU joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Hardware
 112: Suggested that MEDOID: X.509CertificateData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#CertificateData
 113: Suggested that MEDOID: SmartCard joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Credential
 114: Suggested that MEDOID: HWDeveloper joins cluster: http://www.ida.liu.se/~almhe/Security.owl#OfficeUser
 115: Suggested that MEDOID: RSA joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Public-KeyEncryption
 116: Suggested that MEDOID: Retina joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential
 117: Suggested that MEDOID: ChecksumData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Credential
 118: Suggested that MEDOID: HostOnIntranet joins cluster: http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost
 119: Suggested that MEDOID: BufferOverflow joins cluster: http://www.ida.liu.se/~almhe/Security.owl#MalformedInput
 120: Suggested that MEDOID: RoutingTablePoisoningThroughLink joins cluster: http://www.ida.liu.se/~almhe/Security.owl#RoutingTablePoisoning
 121: Suggested that MEDOID: Encryption joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Encryption
 122: Suggested that MEDOID: Goal joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 123: Suggested that MEDOID: BruteForceAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BruteForceAttack
 124: Suggested that MEDOID: Steganography joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Cryptography
 125: Suggested that MEDOID: Heap joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DataOnVolatileMedia
 126: Suggested that MEDOID: CBC-MAC joins cluster: http://www.ida.liu.se/~almhe/Security.owl#MACAlgorithm
 127: Suggested that MEDOID: DataInTransit joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Data
 128: Suggested that MEDOID: HostOnWirelessNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Host
 129: Suggested that MEDOID: BypassingIntendedControls joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BypassingIntendedControls
 130: Suggested that MEDOID: ThreatThreatensGoalOfAsset joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasGoal
 131: Suggested that MEDOID: DoS-TCP joins cluster: http://www.ida.liu.se/~almhe/Security.owl#InternetInfrastructureDenialOfService
 132: Suggested that MEDOID: HasThreat joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasThreat
 133: Suggested that MEDOID: NetworkLayerPacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DataInTransit
 134: Suggested that MEDOID: Host-BasedIntrusionDetectionSystem joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ByAuditSourceLocation
 135: Suggested that MEDOID: MD5 joins cluster: http://www.ida.liu.se/~almhe/Security.owl#CryptographicHashFunction
 136: Suggested that MEDOID: SSH joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 137: Suggested that MEDOID: Unix joins cluster: http://www.ida.liu.se/~almhe/Security.owl#OperatingSystem
 138: Suggested that MEDOID: PhysicalBiometricRecognition joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BiometricAuthenticationSystem
 139: Suggested that MEDOID: RC4 joins cluster: http://www.ida.liu.se/~almhe/Security.owl#StreamCipher
 140: Suggested that MEDOID: PacketMistreatment joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PacketMistreatment
 141: Suggested that MEDOID: _3TDES joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TripleDES
 142: Suggested that MEDOID: HandMeasurement joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential
 143: Suggested that MEDOID: Disruption joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Disruption
 144: Suggested that MEDOID: ProgramSourceCodeFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#File
 145: Suggested that MEDOID: FileAccessControl joins cluster: http://www.ida.liu.se/~almhe/Security.owl#FileAccessControl
 146: Suggested that MEDOID: HTMLScriptInjection joins cluster: http://www.ida.liu.se/~almhe/Security.owl#CodeInjection
 147: Suggested that MEDOID: VulnerabilityInCode joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Vulnerability
 148: Suggested that MEDOID: DigitalSignatureData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#MessageDigestData

149: Suggested that MEDOID: PolicyMaker joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TrustManagement
 150: Suggested that MEDOID: VoiceRecognition joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BehaviouralBiometricRecognition
 151: Suggested that MEDOID: OperatingSystem joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Product
 152: Suggested that MEDOID: IPsec joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 153: Suggested that MEDOID: ManInTheMiddle joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Impersonation
 154: Suggested that MEDOID: PrivateKey joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Data
 155: Suggested that MEDOID: Auditing joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Monitoring
 156: Suggested that MEDOID: PrivateUser joins cluster: http://www.ida.liu.se/~almhe/Security.owl#User
 157: Suggested that MEDOID: SourceRoutingAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BypassingIntendedControls
 158: Suggested that MEDOID: GaitRecognition joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BehaviouralBiometricRecognition
 159: Suggested that MEDOID: Hardware joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Technology
 160: Suggested that MEDOID: ProgramFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#File
 161: Suggested that MEDOID: HostOnIntranet joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HostOnIntranet
 162: Suggested that MEDOID: ChecksumData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Credential
 163: Suggested that MEDOID: OfficeUser joins cluster: http://www.ida.liu.se/~almhe/Security.owl#OfficeUser
 164: Suggested that MEDOID: MessageAuthenticationCodeData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DigitalSignatureData
 165: Suggested that MEDOID: MissingErrorHandler joins cluster: http://www.ida.liu.se/~almhe/Security.owl#VulnerabilityInCode
 166: Suggested that MEDOID: Database joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Product
 167: Suggested that MEDOID: DistributedDenialOfService joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DenialOfService
 168: Suggested that MEDOID: One-TimePassword joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Password
 169: Suggested that MEDOID: SQLInjection joins cluster: http://www.ida.liu.se/~almhe/Security.owl#CodeInjection
 170: Suggested that MEDOID: InsecureDefaultSettingUnchanged joins cluster: http://www.ida.liu.se/~almhe/Security.owl#InsecureDefaultSettingUnchanged
 171: Suggested that MEDOID: Compartmentalisation joins cluster: http://www.ida.liu.se/~almhe/Security.owl#AccessControlModel
 172: Suggested that MEDOID: EncryptionKey joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Credential
 173: Suggested that MEDOID: ApplicationLevelGateway joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Firewall
 174: Suggested that MEDOID: SigningWithX.509Certificate joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SigningWithCertificate
 175: Suggested that MEDOID: KerberosArchitecture joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SingleSignOnSystem
 176: Suggested that MEDOID: Gait joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BehaviouralBiometricCredential
 177: Suggested that MEDOID: Debugger joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SecurityHardware
 178: Suggested that MEDOID: DataOnVolatileMedia joins cluster: http://www.ida.liu.se/~almhe/Security.owl#StationaryData
 179: Suggested that MEDOID: PhysicalBiometricCredential joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BiometricCredential
 180: Suggested that MEDOID: PolicyCompliance joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasGoal
 181: Suggested that MEDOID: MessageDigestData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Credential
 182: Suggested that MEDOID: SourceCodeAnalysis joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Countermeasure
 183: Suggested that MEDOID: LocalHost joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HostOnIntranet
 184: Suggested that MEDOID: PrivateKey joins cluster: http://www.ida.liu.se/~almhe/Security.owl#EncryptionKey
 185: Suggested that MEDOID: UnconnectedHost joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Host
 186: Suggested that MEDOID: IntrusionDetectionSystem joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Monitoring
 187: Suggested that MEDOID: SignatureAlgorithm joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Cryptography
 188: Suggested that MEDOID: Intranet joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Network
 189: Suggested that MEDOID: RoutingTablePoisoning joins cluster: http://www.ida.liu.se/~almhe/Security.owl#RoutingTablePoisoning
 190: Suggested that MEDOID: CodeInjection joins cluster: http://www.ida.liu.se/~almhe/Security.owl#MalformedInput
 191: Suggested that MEDOID: CertificateData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#CertificateData
 192: Suggested that MEDOID: Authenticity joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasGoal
 193: Suggested that MEDOID: ChecksumAlgorithm joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Cryptography
 194: Suggested that MEDOID: AuthenticationGoals joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 195: Suggested that MEDOID: HostOnWireNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Host
 196: Suggested that MEDOID: VulnerabilityInConfiguration joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Vulnerability
 197: Suggested that MEDOID: PacketFilter joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PacketFilter
 198: Suggested that MEDOID: Fingerprint joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential

199: Suggested that MEDOID: VulnerabilityInUse joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Vulnerability>

200: Suggested that MEDOID: _TrustedNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Network

201: Suggested that MEDOID: TrustedNetwork joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Network>

202: Suggested that MEDOID: User joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#User>

203: Suggested that MEDOID: PSP joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Standard>

204: Suggested that MEDOID: _Deflection joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasMechanismType>

205: Suggested that MEDOID: Bacteria joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Virus>

206: Suggested that MEDOID: E-Mail joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#ApplicationLayerPacket>

207: Suggested that MEDOID: DenialOfService joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DenialOfService>

208: Suggested that MEDOID: _BackupFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_File

209: Suggested that MEDOID: DNSHacking joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DNSHacking>

210: Suggested that MEDOID: _DatabaseDataFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_File

211: Suggested that MEDOID: Anti-VirusSoftware joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Monitoring>

212: Suggested that MEDOID: _Prevention joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasMechanismType>

213: Suggested that MEDOID: FacialPatternRecognition joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricRecognition>

214: Suggested that MEDOID: Firewall joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Monitoring>

215: Suggested that MEDOID: Literature joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Literature>

216: Suggested that MEDOID: VPN joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SecureNetworkCommunication>

217: Suggested that MEDOID: Backup joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Countermeasure>

218: Suggested that MEDOID: JavaSandbox joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#AccessControlMechanism>

219: Suggested that MEDOID: Remailer joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SecureNetworkCommunication>

220: Suggested that MEDOID: _UDPpacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_TransportLayerPacket

221: Suggested that MEDOID: ConfidentialityGoals joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Goal>

222: Suggested that MEDOID: Threat joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Threat>

223: Suggested that MEDOID: ByAuditSourceLocation joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#IntrusionDetectionSystem>

224: Suggested that MEDOID: FacialPattern joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential>

225: Suggested that MEDOID: _E-Mail joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_DataInTransit

226: Suggested that MEDOID: Windows joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#OperatingSystem>

227: Suggested that MEDOID: PeriodicIntrusionDetectionSystem joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#ByUsageFrequency>

228: Suggested that MEDOID: SHA-1 joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#CryptographicHashFunction>

229: Suggested that MEDOID: S-HTTP joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Standard>

230: Suggested that MEDOID: _TransportLayerPacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_DataInTransit

231: Suggested that MEDOID: Router joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Router>

232: Suggested that MEDOID: TrafficAnalysis joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Eavesdropping>

233: Suggested that MEDOID: Disclosure joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Threat>

234: Suggested that MEDOID: Standard joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Standard>

235: Suggested that MEDOID: ServerHost joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost>

236: Suggested that MEDOID: TCPpacket joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#TransportLayerPacket>

237: Suggested that MEDOID: DenialOfService joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DNSHacking>

238: Suggested that MEDOID: FingerprintRecognition joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricRecognition>

239: Suggested that MEDOID: Password joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Credential>

240: Suggested that MEDOID: RadiationMonitoring joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SideChannelAttack>

241: Suggested that MEDOID: HTTPS joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Standard>

242: Suggested that MEDOID: ByUsageFrequency joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#IntrusionDetectionSystem>

243: Suggested that MEDOID: EmissionSecurity joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Countermeasure>

244: Suggested that MEDOID: BastionHost joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost>

245: Suggested that MEDOID: _Data joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Technology

246: Suggested that MEDOID: EncryptionHardware joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SecurityHardware>

247: Suggested that MEDOID: MemoryProtection joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Countermeasure>

248: Suggested that MEDOID: IPAddressSpoofing joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Impersonation>

249: Suggested that MEDOID: TrafficPadding joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SecureNetworkCommunication>

250: Suggested that MEDOID: PhysicalBiometricCredential joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential>

251: Suggested that MEDOID: SWDeveloper joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#OfficeUser>

252: Suggested that MEDOID: RIPE-MAC joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#MACAlgorithm>

253: Suggested that MEDOID: StatisticalAttack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PassiveAttack>

254: Suggested that MEDOID: SecureNetworkCommunication joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SecureNetworkCommunication>

255: Suggested that MEDOID: HeapOverflow joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#BufferOverflow>

256: Suggested that MEDOID: _PublicKey joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_EncryptionKey

257: Suggested that MEDOID: Theft joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PhysicalThreat>

258: Suggested that MEDOID: MisuseDetectionSystem joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#ByDetectionMethod>

259: Suggested that MEDOID: Asset joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Asset>

260: Suggested that MEDOID: BehaviouralBiometricCredential joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#BehaviouralBiometricCredential>

261: Suggested that MEDOID: _Voice joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_BehaviouralBiometricCredential

262: Suggested that MEDOID: _Host joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Technology

263: Suggested that MEDOID: HostOnWirelessNetwork joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#NetworkedHost>

264: Suggested that MEDOID: SymmetricAlgorithm joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#SymmetricAlgorithm>

265: Suggested that MEDOID: HasAsset joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasAsset>

266: Suggested that MEDOID: _Correctness joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasGoal>

267: Suggested that MEDOID: AccessControlModel joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Model>

268: Suggested that MEDOID: RoutingTablePoisoningThroughRouter joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#RoutingTablePoisoning>

269: Suggested that MEDOID: PublicKey joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Data>

270: Suggested that MEDOID: CrossSiteScripting joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#CodeInjection>

271: Suggested that MEDOID: _Dongle joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Credential

272: Suggested that MEDOID: X.509 joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Standard>

273: Suggested that MEDOID: DatabaseDataFile joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#File>

274: Suggested that MEDOID: Impersonation joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Impersonation>

275: Suggested that MEDOID: HasCountermeasure joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasCountermeasure>

276: Suggested that MEDOID: InternetInfrastructureAttack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#InternetInfrastructureAttack>

277: Suggested that MEDOID: _HostOnInternet joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Host

278: Suggested that MEDOID: CircuitLevelGateway joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Firewall>

279: Suggested that MEDOID: LogMonitoring joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Monitoring>

280: Suggested that MEDOID: Knowledge-BasedIntrusionDetectionSystem joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#ByDetectionMethod>

281: Suggested that MEDOID: SideChannelAttack joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#PassiveAttack>

282: Suggested that MEDOID: TimeOfEmployment joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#TimeOfEmployment>

283: Suggested that MEDOID: _HTTPData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_ApplicationLayerPacket

284: Suggested that MEDOID: MS_Word joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Editor>

285: Suggested that MEDOID: LocalHost joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HostOnIntranet>

286: Suggested that MEDOID: PhysicalThreat joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Disruption>

287: Suggested that MEDOID: _Sender joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Human

288: Suggested that MEDOID: _Authentication joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasGoal>

289: Suggested that MEDOID: DBAccessControl joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#FileAccessControl>

290: Suggested that MEDOID: PassiveIntrusionDetectionSystem joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#ByBehaviourOnDetection>

291: Suggested that MEDOID: _ServerHost joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Host

292: Suggested that MEDOID: AccessControlMechanism joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#AccessControlMechanism>

293: Suggested that MEDOID: _ApplicationLayerPacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_DataInTransit

294: Suggested that MEDOID: Dongle joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Credential>

295: Suggested that MEDOID: BiometricCredential joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#Credential>

296: Suggested that MEDOID: _Accountability joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasGoal>

297: Suggested that MEDOID: _Non-Repudiation joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#HasGoal>

298: Suggested that MEDOID: NetworkLayerPacket joins cluster: <http://www.ida.liu.se/~almhe/Security.owl#DataInTransit>

299: Suggested that MEDOID: DefaultPasswordUnchanged joins cluster: http://www.ida.liu.se/~almhe/Security.owl#InsecureDefaultSettingUnchanged
 300: Suggested that MEDOID: DiffieHellman joins cluster: http://www.ida.liu.se/~almhe/Security.owl#KeyExchange
 301: Suggested that MEDOID: _File joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DataOnNon-VolatileMedia
 302: Suggested that MEDOID: TripleDES joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SymmetricAlgorithm
 303: Suggested that MEDOID: Masquerading joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Masquerading
 304: Suggested that MEDOID: WindowAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DenialOfService
 305: Suggested that MEDOID: PasswordSniffing joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Eavesdropping
 306: Suggested that MEDOID: Backdoor joins cluster: http://www.ida.liu.se/~almhe/Security.owl#MaliciousCode
 307: Suggested that MEDOID: Virus joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Virus
 308: Suggested that MEDOID: UDPPacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TransportLayerPacket
 309: Suggested that MEDOID: Overclocking joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ActiveHardwareMisuse
 310: Suggested that MEDOID: _Process joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Technology
 311: Suggested that MEDOID: PacketMistreatmentRouter joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PacketMistreatment
 312: Suggested that MEDOID: IntegerOverflow joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BufferOverflow
 313: Suggested that MEDOID: TimeOfCheckToTimeOfUse joins cluster: http://www.ida.liu.se/~almhe/Security.owl#RaceCondition
 314: Suggested that MEDOID: Rabbit-Worm joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Worm
 315: Suggested that MEDOID: _Secrecy joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ConfidentialitySimilar
 316: Suggested that MEDOID: HasMechanismType joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasMechanismType
 317: Suggested that MEDOID: ElGamalSignatureScheme joins cluster: http://www.ida.liu.se/~almhe/Security.owl#CryptographicHashFunction
 318: Suggested that MEDOID: Firefly joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 319: Suggested that MEDOID: ProgramSourceCodeFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#File
 320: Suggested that MEDOID: NaryRelation joins cluster: http://www.ida.liu.se/~almhe/Security.owl#NaryRelation
 321: Suggested that MEDOID: PowerMonitoringAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SideChannelAttack
 322: Suggested that MEDOID: AdHocNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Network
 323: Suggested that MEDOID: _WirelessNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Network
 324: Suggested that MEDOID: Eavesdropping joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PassiveAttack
 325: Suggested that MEDOID: HandMeasurementRecognition joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricRecognition
 326: Suggested that MEDOID: IntegrityGoals joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 327: Suggested that MEDOID: FacialPattern joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_PhysicalBiometricCredential
 328: Suggested that MEDOID: StatefulInspectionPacketFilter joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PacketFilter
 329: Suggested that MEDOID: Heat joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalThreat
 330: Suggested that MEDOID: Flooding joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DenialOfService
 331: Suggested that MEDOID: Keylogger joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Spyware
 332: Suggested that MEDOID: ConfigurationFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#File
 333: Suggested that MEDOID: _Identification joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasGoal
 334: Suggested that MEDOID: Voice joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BehaviouralBiometricCredential
 335: Suggested that MEDOID: SingleSignOnSystem joins cluster: http://www.ida.liu.se/~almhe/Security.owl#LoginSystem
 336: Suggested that MEDOID: _WiredNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Network
 337: Suggested that MEDOID: TechnologyMistakenlyUsedAsCountermeasure joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TechnologyMistakenlyUsedAsCountermeasure
 338: Suggested that MEDOID: BehaviouralBiometricRecognition joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BiometricAuthenticationSystem
 339: Suggested that MEDOID: _WirelessAccessPoint joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Router
 340: Suggested that MEDOID: CryptographicHashFunction joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SignatureAlgorithm
 341: Suggested that MEDOID: TrustManagement joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TrustManagement
 342: Suggested that MEDOID: TimingAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SideChannelAttack
 343: Suggested that MEDOID: _Network joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Technology
 344: Suggested that MEDOID: AES joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 345: Suggested that MEDOID: Host joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Technology
 346: Suggested that MEDOID: AdHocNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Network
 347: Suggested that MEDOID: CachePoisoning joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DNSHacking

348: Suggested that MEDOID: ByDetectionMethod joins cluster: http://www.ida.liu.se/~almhe/Security.owl#IntrusionDetectionSystem
 349: Suggested that MEDOID: DMSServerCompromising joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DNSHacking
 350: Suggested that MEDOID: SmartCard joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Credential
 351: Suggested that MEDOID: GuessingAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BruteForceAttack
 352: Suggested that MEDOID: TargetConnectedToNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Vulnerability
 353: Suggested that MEDOID: Logging joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Monitoring
 354: Suggested that MEDOID: EllipticCurveCryptography joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Public-KeyEncryption
 355: Suggested that MEDOID: LoginName joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Credential
 356: Suggested that MEDOID: S-MIME joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 357: Suggested that MEDOID: Sender joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Human
 358: Suggested that MEDOID: MaliciousCode joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Impersonation
 359: Suggested that MEDOID: Spoofing joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Spoofing
 360: Suggested that MEDOID: BackupFile joins cluster: http://www.ida.liu.se/~almhe/Security.owl#File
 361: Suggested that MEDOID: _Privacy joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ConfidentialitySimilar
 362: Suggested that MEDOID: EllipticCurveDiscretelogarithm joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Discretelogarithm
 363: Suggested that MEDOID: _Deterrence joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasMechanismType
 364: Suggested that MEDOID: SystemModification joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ActiveAttack
 365: Suggested that MEDOID: _UntrustedNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Network
 366: Suggested that MEDOID: Iris joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential
 367: Suggested that MEDOID: HoneyPot joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TechnologyMistakenlyUsedAsCountermeasure
 368: Suggested that MEDOID: ActiveHardwareMisuse joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Disruption
 369: Suggested that MEDOID: DictionaryAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#GuessingAttack
 370: Suggested that MEDOID: OpenPGP joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 371: Suggested that MEDOID: Usurpation joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Usurpation
 372: Suggested that MEDOID: PingOfDeath joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BufferOverflow
 373: Suggested that MEDOID: MacOS joins cluster: http://www.ida.liu.se/~almhe/Security.owl#OperatingSystem
 374: Suggested that MEDOID: IrisRecognition joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricRecognition
 375: Suggested that MEDOID: Deception joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Deception
 376: Suggested that MEDOID: Editor joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Product
 377: Suggested that MEDOID: FormatStringMisuse joins cluster: http://www.ida.liu.se/~almhe/Security.owl#MisuseOfLanguageConstruct
 378: Suggested that MEDOID: BlockCipher joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BlockCipher
 379: Suggested that MEDOID: VulnerableServiceStartsAutomatically joins cluster: http://www.ida.liu.se/~almhe/Security.owl#InsecureDefaultSettingUnchanged
 380: Suggested that MEDOID: _DigitalSignatureData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_ChecksumData
 381: Suggested that MEDOID: Rabbit joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Virus
 382: Suggested that MEDOID: _TCPacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_TransportLayerPacket
 383: Suggested that MEDOID: MessageAuthenticationCode joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Checksum
 384: Suggested that MEDOID: DSA joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 385: Suggested that MEDOID: DES joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 386: Suggested that MEDOID: ByBehaviourOnDetection joins cluster: http://www.ida.liu.se/~almhe/Security.owl#IntrusionDetectionSystem
 387: Suggested that MEDOID: MissingInputValidation joins cluster: http://www.ida.liu.se/~almhe/Security.owl#VulnerabilityInCode
 388: Suggested that MEDOID: ActiveAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Threat
 389: Suggested that MEDOID: _Gait joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BehaviouralBiometricCredential
 390: Suggested that MEDOID: PacketMistreatmentLink joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PacketMistreatment
 391: Suggested that MEDOID: Confidentiality joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ConfidentialitySimilar
 392: Suggested that MEDOID: MisuseOfLanguageConstruct joins cluster: http://www.ida.liu.se/~almhe/Security.owl#MisuseOfLanguageConstruct
 393: Suggested that MEDOID: TinyFragmentAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BypassingIntendedControls
 394: Suggested that MEDOID: _EncryptionHardware joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_SecurityHardware
 395: Suggested that MEDOID: Phishing joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Impersonation
 396: Suggested that MEDOID: EncryptionAlgorithm joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Cryptography

397: Suggested that MEDOID: IPSplicing joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SessionHijacking
 398: Suggested that MEDOID: Behaviour-BasedIntrusionDetectionSystem joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ByDetectionMethod
 399: Suggested that MEDOID: StationaryData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Data
 400: Suggested that MEDOID: Fingerprint joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential
 401: Suggested that MEDOID: EmailFilter joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Monitoring
 402: Suggested that MEDOID: ShellInjection joins cluster: http://www.ida.liu.se/~almhe/Security.owl#CodeInjection
 403: Suggested that MEDOID: CountermeasureProtectsGoalOfAssetThroughType joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasGoal
 404: Suggested that MEDOID: HTTPData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ApplicationLayerPacket
 405: Suggested that MEDOID: DoS-ICMP joins cluster: http://www.ida.liu.se/~almhe/Security.owl#InternetInfrastructureDenialOfService
 406: Suggested that MEDOID: HostCompromising joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Disruption
 407: Suggested that MEDOID: RIPE-MD-160 joins cluster: http://www.ida.liu.se/~almhe/Security.owl#CryptographicHashFunction
 408: Suggested that MEDOID: TransportLayerPacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DataInTransit
 409: Suggested that MEDOID: _IPPacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#NetworkLayerPacket
 410: Suggested that MEDOID: Network-BasedIntrusionDetectionSystem joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ByAuditSourceLocation
 411: Suggested that MEDOID: SessionHijacking joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Impersonation
 412: Suggested that MEDOID: WirelessNetwork joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Network
 413: Suggested that MEDOID: Countermeasure joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Countermeasure
 414: Suggested that MEDOID: DBMS joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Product
 415: Suggested that MEDOID: SwitchingOffFan joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Heat
 416: Suggested that MEDOID: HandMeasurement joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential
 417: Suggested that MEDOID: NAKAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#ActiveAttack
 418: Suggested that MEDOID: SAML joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 419: Suggested that MEDOID: SystemAdministrator joins cluster: http://www.ida.liu.se/~almhe/Security.owl#OfficelUser
 420: Suggested that MEDOID: Retina joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricCredential
 421: Suggested that MEDOID: Degausser joins cluster: http://www.ida.liu.se/~almhe/Security.owl#SecurityHardware
 422: Suggested that MEDOID: _BehaviouralBiometricCredential joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_BiometricCredential
 423: Suggested that MEDOID: _NetworkedHost joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Host
 424: Suggested that MEDOID: SymmetricKey joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Data
 425: Suggested that MEDOID: _Harddisk joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Hardware
 426: Suggested that MEDOID: RetinaRecognition joins cluster: http://www.ida.liu.se/~almhe/Security.owl#PhysicalBiometricRecognition
 427: Suggested that MEDOID: _SymmetricKey joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_EncryptionKey
 428: Suggested that MEDOID: Replay joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Impersonation
 429: Suggested that MEDOID: ApplicationLayerPacket joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DataInTransit
 430: Suggested that MEDOID: TDES-EDE joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TripleDES
 431: Suggested that MEDOID: ElGamalAlgorithm joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Public-KeyEncryption
 432: Suggested that MEDOID: Process joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Technology
 433: Suggested that MEDOID: CryptographyModel joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Model
 434: Suggested that MEDOID: _Intranet joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Network
 435: Suggested that MEDOID: CodeSigning joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DigitalSigning
 436: Suggested that MEDOID: KerberosAuthenticationProtocol joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Standard
 437: Suggested that MEDOID: _Password joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_Credential
 438: Suggested that MEDOID: UnconnectedHost joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Host
 439: Suggested that MEDOID: _MessageAuthenticationCodeData joins cluster: http://www.ida.liu.se/~almhe/Security.owl#_DigitalSignatureData
 440: Suggested that MEDOID: Public-KeyEncryption joins cluster: http://www.ida.liu.se/~almhe/Security.owl#EncryptionAlgorithm
 441: Suggested that MEDOID: ScanningAttack joins cluster: http://www.ida.liu.se/~almhe/Security.owl#BruteForceAttack
 442: Suggested that MEDOID: _Correction joins cluster: http://www.ida.liu.se/~almhe/Security.owl#HasMechanismType
 443: Suggested that MEDOID: File joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DataOnNon-VolatileMedia
 444: Suggested that MEDOID: VulnerabilityScanner joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Countermeasure
 445: Suggested that MEDOID: DataOnVolatileMedia joins cluster: http://www.ida.liu.se/~almhe/Security.owl#StationaryData
 446: Suggested that MEDOID: Authentication joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal

447: Suggested that MEDOID: Anonymity joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 448: Suggested that MEDOID: Correctness joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 449: Suggested that MEDOID: AC-CIIT joins cluster: http://www.ida.liu.se/~almhe/SecurityLiterature.owl#Definition
 450: Suggested that MEDOID: Integrity joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 451: Suggested that MEDOID: Authenticity joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 452: Suggested that MEDOID: Detection joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DefenseStrategy
 453: Suggested that MEDOID: AtLoadTime joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TimeOfEmployment
 454: Suggested that MEDOID: Correction joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DefenseStrategy
 455: Suggested that MEDOID: CCIT joins cluster: ***** No Cluster Suggested. *****
 456: Suggested that MEDOID: Identification joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 457: Suggested that MEDOID: Confidentiality joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 458: Suggested that MEDOID: Secrecy joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 459: Suggested that MEDOID: AtCompileTime joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TimeOfEmployment
 460: Suggested that MEDOID: Deterrence joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DefenseStrategy
 461: Suggested that MEDOID: Security.owl joins cluster: ***** No Cluster Suggested. *****
 462: Suggested that MEDOID: Trust joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 463: Suggested that MEDOID: Sch96 joins cluster: http://www.ida.liu.se/~almhe/SecurityLiterature.owl#Literature
 464: Suggested that MEDOID: AtRuntime joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TimeOfEmployment
 465: Suggested that MEDOID: Prevention joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DefenseStrategy
 466: Suggested that MEDOID: Periodically joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TimeOfEmployment
 467: Suggested that MEDOID: AtWriteTime joins cluster: http://www.ida.liu.se/~almhe/Security.owl#TimeOfEmployment
 468: Suggested that MEDOID: Non-Repudiation joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 469: Suggested that MEDOID: KLK05 joins cluster: http://www.ida.liu.se/~almhe/SecurityLiterature.owl#Literature
 470: Suggested that MEDOID: Recovery joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DefenseStrategy
 471: Suggested that MEDOID: Accountability joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 472: Suggested that MEDOID: Deflection joins cluster: http://www.ida.liu.se/~almhe/Security.owl#DefenseStrategy
 473: Suggested that MEDOID: Authorisation joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 474: Suggested that MEDOID: PolicyCompliance joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 475: Suggested that MEDOID: Availability joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal
 476: Suggested that MEDOID: Privacy joins cluster: http://www.ida.liu.se/~almhe/Security.owl#Goal

Appendix C

POLICY_1.drl

Page 1 of 1

```
1: package ie.tcd.cs.kdeg.mecon.DR00LS.RULES;
2:
3: import ie.tcd.cs.kdeg.mecon.DR00LS.MIBS.Subscriber;
4: import ie.tcd.cs.kdeg.mecon.DR00LS.RULES.Actioner;
5: import ie.tcd.cs.kdeg.mecon.Cluster_Mgmt.ClusterCreator;
6:
7:
8: rule "Where should Subscribers be placed (within the network) in the first Instance."
9:
10:  when
11:
12:      actioner : Actioner();
13:      sub : Subscriber();
14:
15:      clusCrea : ClusterCreator(brokersSize >= 2,
16:                               brokersRunning == true);
17:
18:  then
19:
20:      actioner.performSubscrClustering(sub.getmaster(), sub.getsubscriberID() , sub.getmedoid());
21:
22:  end
23:
```

POLICY_2.drl

Page 1 of 1

```
1: package ie.tcd.cs.kdeg.mecon.DR00LS.RULES;
2:
3: import ie.tcd.cs.kdeg.mecon.Cluster_Mgmt.ClusterCreator;
4: import ie.tcd.cs.kdeg.mecon.DR00LS.RULES.Actioner;
5:
6: rule "When should a publisher push in their Medoids."
7:
8:  when
9:
10:      actioner : Actioner();
11:      clusCrea : ClusterCreator(brokersSize >= 0,
12:                               brokersRunning == true);
13:
14:  then
15:
16:      actioner.requestBrokerMedoids("tcp:127.0.0.1:2002", 2);
17:
18:  end
```

```
1: package ie.tcd.cs.kdeg.mecon.DR00LS.RULES;
2:
3: import ie.tcd.cs.kdeg.mecon.Cluster_Mgmt.ClusterCreator;
4: import ie.tcd.cs.kdeg.mecon.DR00LS.RULES.Actioner;
5: import ie.tcd.cs.kdeg.mecon.DR00LS.MIBS.Broker;
6: import java.util.Iterator;
7:
8: rule "Identify the broker on which clustering should occur."
9: when
10: $highestSpread : Broker()
11:   from accumulate{ $bm : Broker(),
12:     init{ Broker highest = new Broker(); },
13:     action{
14:       if ($bm.getSpread() > highest.getSpread()){
15:         highest = $bm;
16:       }
17:     },
18:     result(highest)
19:   }
20:   System.out.println("Broker with the Highest Spread: " + $highestSpread.getBrokerID());
21:   $highestSpread.setHighestSpreadTrue();
22: end
```

```
1: package ie.tcd.cs.kdeg.mecon.DR00LS.RULES;
2:
3: import ie.tcd.cs.kdeg.mecon.DR00LS.RULES.Actioner;
4: import ie.tcd.cs.kdeg.mecon.DR00LS.MIBS.Broker;
5: import java.util.Iterator;
6:
7: rule "Find the Medoid with the highest Number of Subscribers."
8: when
9:   actioner : Actioner();
10:  broker : Broker(highestSpreadTrue == true);
11:
12: then
13:
14:   if (broker.getNumberOfSubs1() < broker.getNumberOfSubs2()){
15:     System.out.println("Medoid with the Smallest Number of Subscribers = " + broker.getMedoid1() + " from broker " + broker.getBrokerID());
16:     //actioner.performBrokerClustering(broker.getBrokerID(), broker.getMedoid1());
17:   }else if (broker.getNumberOfSubs2() < broker.getNumberOfSubs1()){
18:     System.out.println("Medoid with the Smallest Number of Subscribers = " + broker.getMedoid2() + " from broker " + broker.getBrokerID());
19:     //actioner.performBrokerClustering(broker.getBrokerID(), broker.getMedoid2());
20:   }else if (broker.getNumberOfSubs2() == broker.getNumberOfSubs1()){
21:     System.out.println("Uh Oh... There are an equal number of Subs for both Medoids!");
22:   }
23: end
24:
```

```
1: package ie.tcd.cs.kdeg.mecon.DR00LS.RULES;
2:
3: import ie.tcd.cs.kdeg.mecon.Cluster_Mgmt.ClusterCreator;
4: import ie.tcd.cs.kdeg.mecon.DR00LS.RULES.Actioner;
5:
6: rule "When should a publisher push in their Medoids."
7:
8:   when
9:     actioner : Actioner();
10:    clusCrea : ClusterCreator(brokersSize >= 0,
11:                             publishersSize >= 1,
12:                             numberOfClusters >=1,
13:                             brokersRunning == true);
14:
15:
16:
17:   then
18:     actioner.requestPublishersMedoids("tcp:mahler.cs.tcd.ie:2002");
19: end
```

```
1: package ie.tcd.cs.kdeg.mecon.DR00LS.RULES;
2:
3: import ie.tcd.cs.kdeg.mecon.DR00LS.RULES.Actioner;
4: import ie.tcd.cs.kdeg.mecon.DR00LS.MIBS.Publisher;
5: import ie.tcd.cs.kdeg.mecon.Cluster_Mgmt.ClusterCreator;
6: import java.util.Iterator;
7:
8: rule "Where should a Publisher be clustered when they report their medoid"
9:
10:   when
11:     actioner : Actioner();
12:     publisher : Publisher();
13:
14:     clusCrea : ClusterCreator(brokersSize >= 0,
15:                               publishersSize >= 1,
16:                               subscribersSize >= 0);
17:
18:
19:   then
20:
21:     actioner.performPublisherClustering(publisher.getpublisherID(), publisher.getmedoid());
22: end
23:
```

```
1: package ie.tcd.cs.kdeg.mecon.DR00LS.RULES;
2:
3: import ie.tcd.cs.kdeg.mecon.DR00LS.MIBS.Subscriber;
4: import ie.tcd.cs.kdeg.mecon.DR00LS.RULES.Actioner;
5:
6: rule "When should notification reports be requested."
7:
8:   when
9:
10:    actioner : Actioner();
11:
12:   then
13:
14:    actioner.requestNotificationReports("tcp:127.0.0.1:50000", 10);
15:
16:
17:   end
```

```
1: package ie.tcd.cs.kdeg.mecon.DR00LS.RULES;
2:
3: import ie.tcd.cs.kdeg.mecon.DR00LS.MIBS.Subscriber;
4: import ie.tcd.cs.kdeg.mecon.DR00LS.RULES.Actioner;
5: import ie.tcd.cs.kdeg.mecon.Cluster_Mgmt.ClusterCreator;
6:
7: rule "Are notifications sourced from too far across the network?"
8:
9:   when
10:
11:    actioner : Actioner();
12:    sub : Subscriber(latestMean > 1.5,
13:                    latestStdDev >= 0.0);
14:    clus : ClusterCreator();
15:
16:   then
17:
18:    System.out.println("Performing Subscriber re-clustering with: ");
19:    System.out.println(sub.getsubscriberID() + " has a StdDev of " + sub.getLatestStdDev() + " and a Mean of " + sub.getLatestMean());
20:    //actioner.performSubscrClustering(sub.getmaster(), sub.getsubscriberID(), sub.getmedoid() );
21:    actioner.testter(sub.getsubscriberID());
22:
23:   end
24:
25: rule "Are notifications **NOT** sourced from too far across the network?"
26:
27:   dialect "java"
28:
29:   when
30:
31:    actioner : Actioner();
32:    sub : Subscriber(latestMean >= 5.0,
33:                    latestStdDev >= 3.0);
34:    clus : ClusterCreator();
35:
36:   then
37:
38:    System.out.println(sub.getsubscriberID() + " has a StdDev of " + sub.getLatestStdDev() + " and a Mean of " + sub.getLatestMean());
39:    System.out.println("No action needs to be taken - Going back to monitoring for Noitfication Reports!");
40:
41:   end
```

Appendix D

A data DVD is provided with this thesis and includes:

1. Copy of this thesis in Adobe Acrobat PDF format.
2. All Evaluation data sets.
3. Publication and Subscription Data sets used.
4. All ontologies (in *.owl format) used in this thesis.
5. Results of clustering algorithm applied to sample ontologies.
6. All policies used in KBNCcluster.
7. Copies of papers published around this thesis.

Appendix E

For a number of experiments the data value presented does not suffer from any change, regardless of the number of times the experiment is run. For example in the experiments regarding the number of hops taken to deliver a message, regardless of how many times the experiment is run the number of hops taken stays exactly the same.

There are some experiments, particularly those involving measuring time, where when run there is a slight variance in the measurement recorded. In addressing this variance in each of these experiments the data point is calculated for 1,3,5,10 and 15 runs. So in the first experiment (1 run) only 1 value is recorded, in the second experiment (3 runs) the mean from 3 experiments is recorded, the same for 5, 10 and 15 experimental runs.

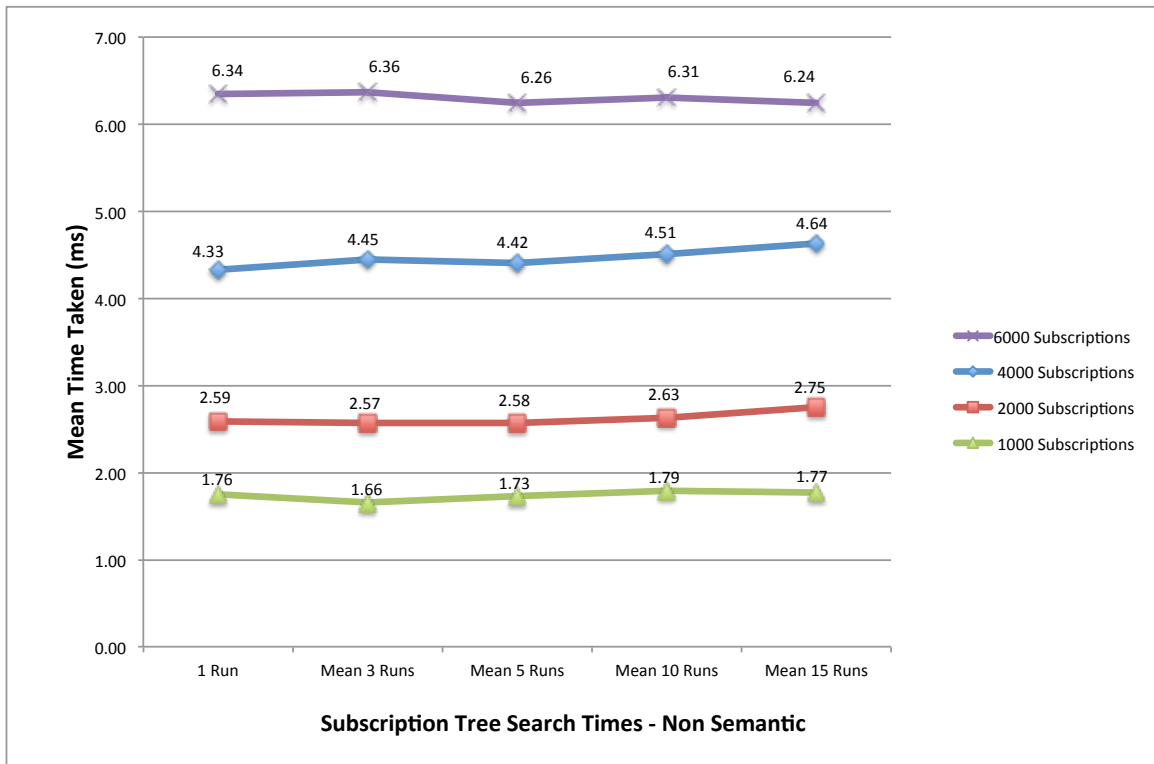
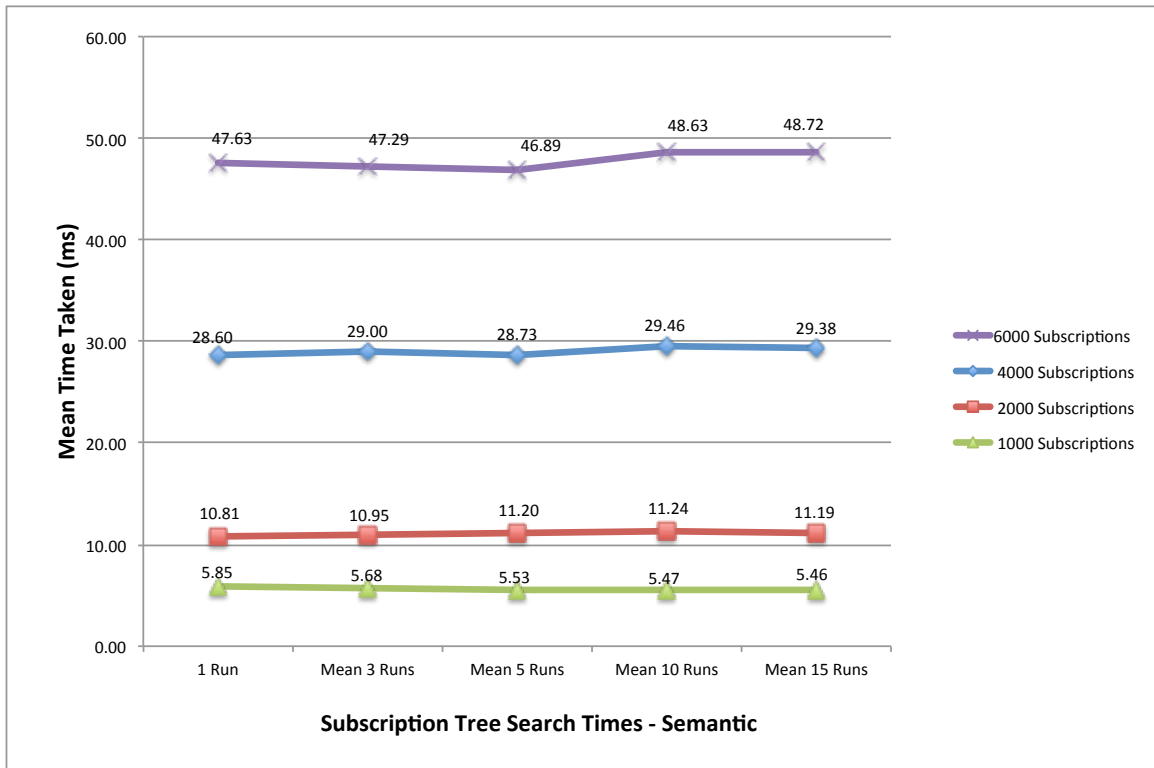
The trend between each of the mean values (1, 3, 5, 10, 15 runs) is compared to establish whether with more experimental data points, the mean value changes i.e. whether as more experimental runs are added to the mean calculation is there a fluctuation in the data values recorded. From this sensitivity analysis the choice of using five mean experimental runs is justified.

In this appendix sensitivity analysis is presented, in graphical form for the following experiments:

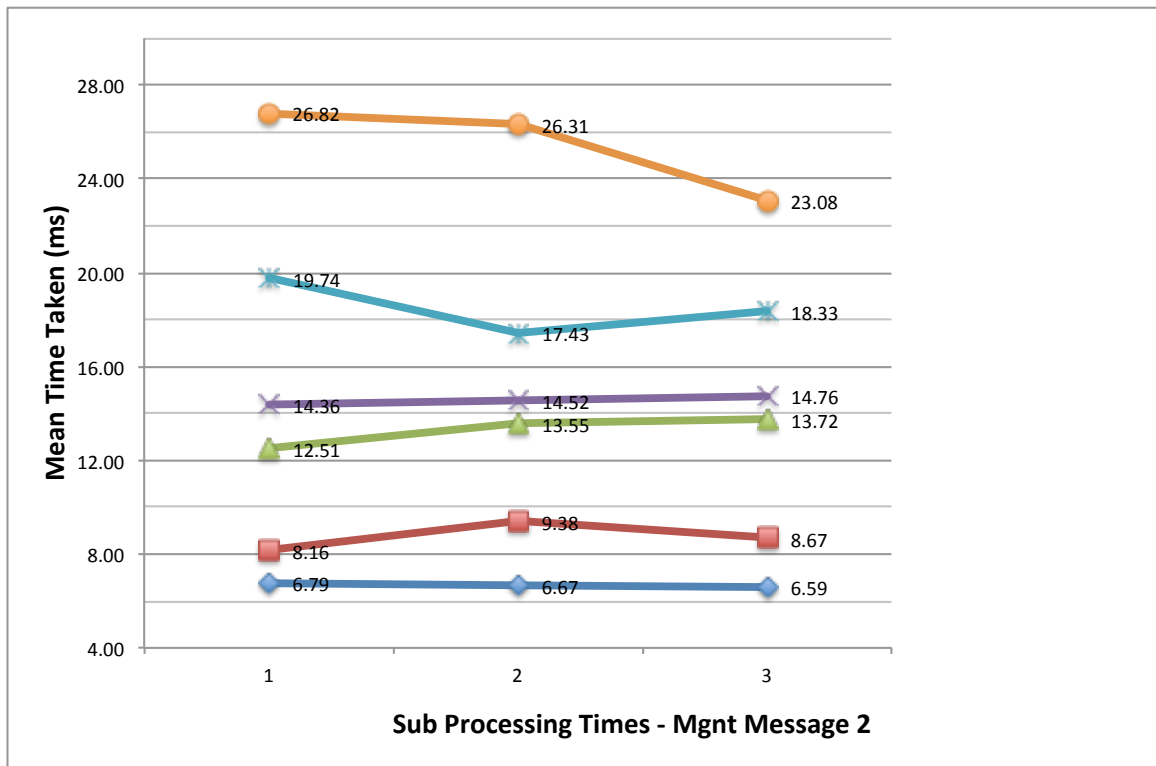
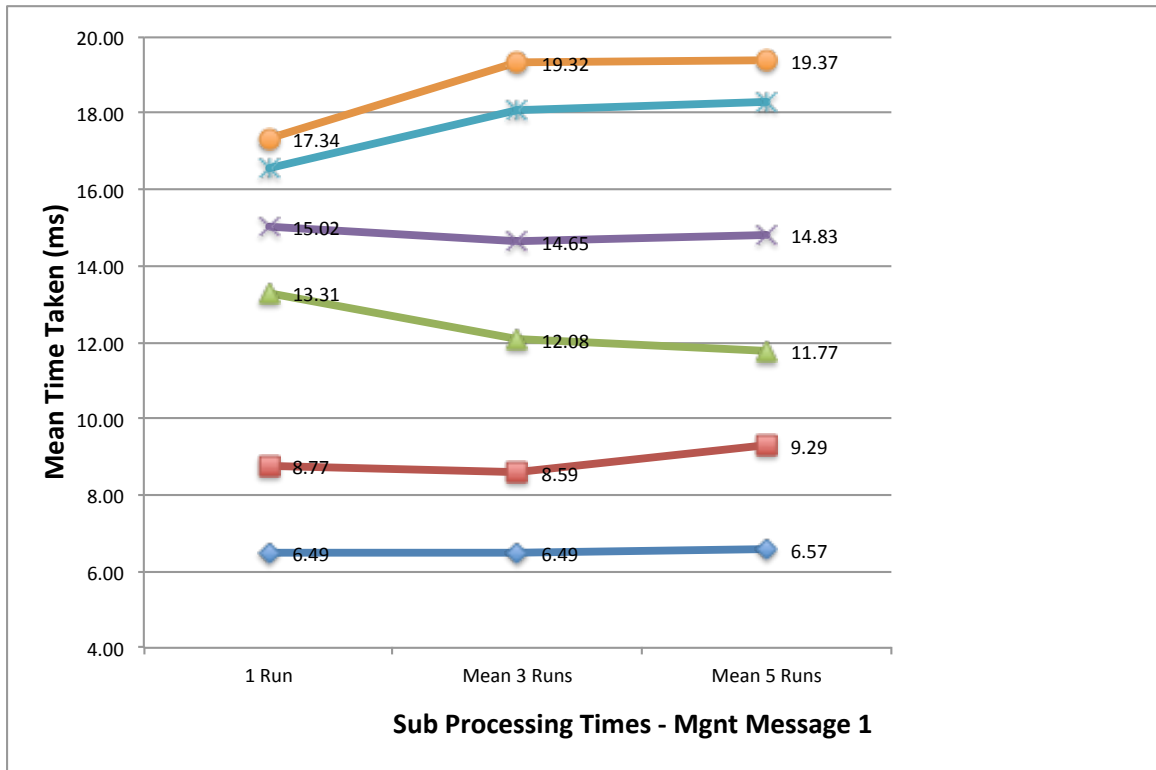
- 6.3.2 Subscription Tree Search Time
- 6.4.2.4 Subscription Processing Times
- 6.4.2.5 Publication Processing Times
- 6.4.2.6 Pub-to-Sub Delivery Times
- 6.4.3.1 Moving Broker & Moving All Subscribers

For experiments (6.4.2.4, 6.4.2.5 and 6.4.2.6) data is only presented for 1, 3 and 5 experimental runs, as these experiments are particularly sensitive to network delays in processing and delivering messages. Additionally due to the unpredictable and high costs involved in management message collection, and as discussed in the evaluation chapter of this thesis the approach taken in these experiments is not used in the final KBNCluster implementation. For the remainder of the experiments, used in the final implementation of KBNCluster, full sensitivity analysis, for 1, 3, 5, 10 and 15 experimental runs, is presented.

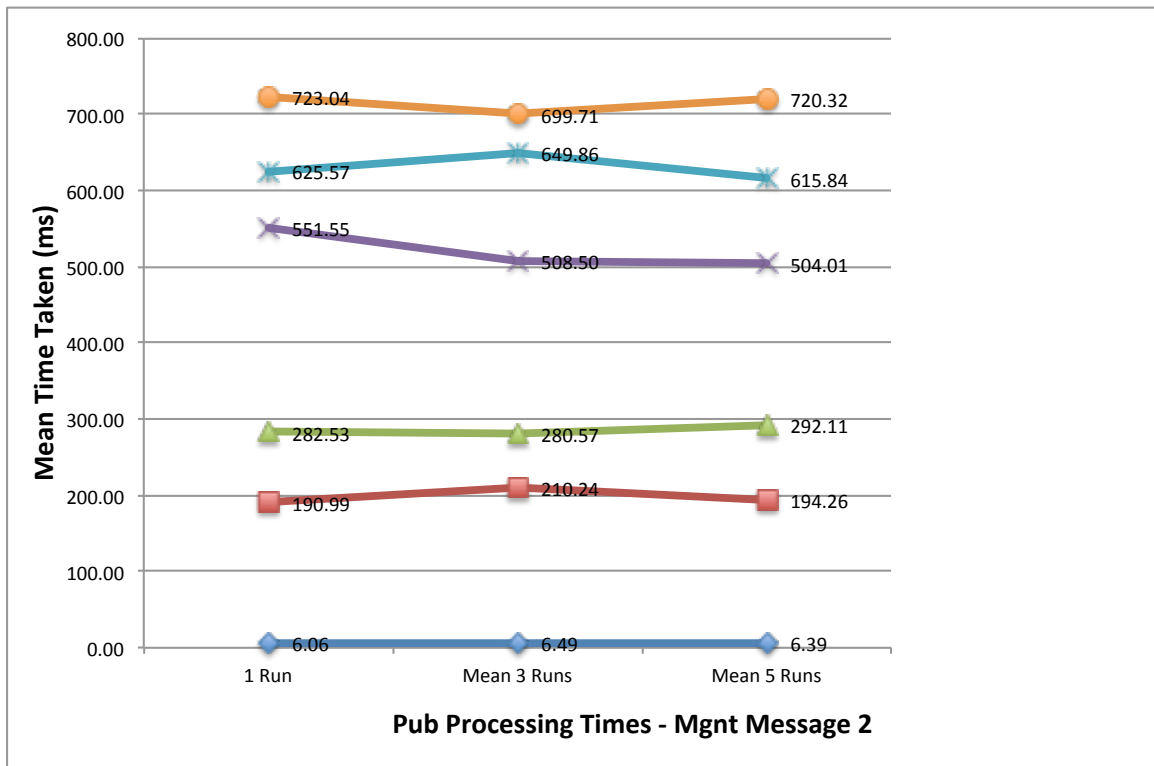
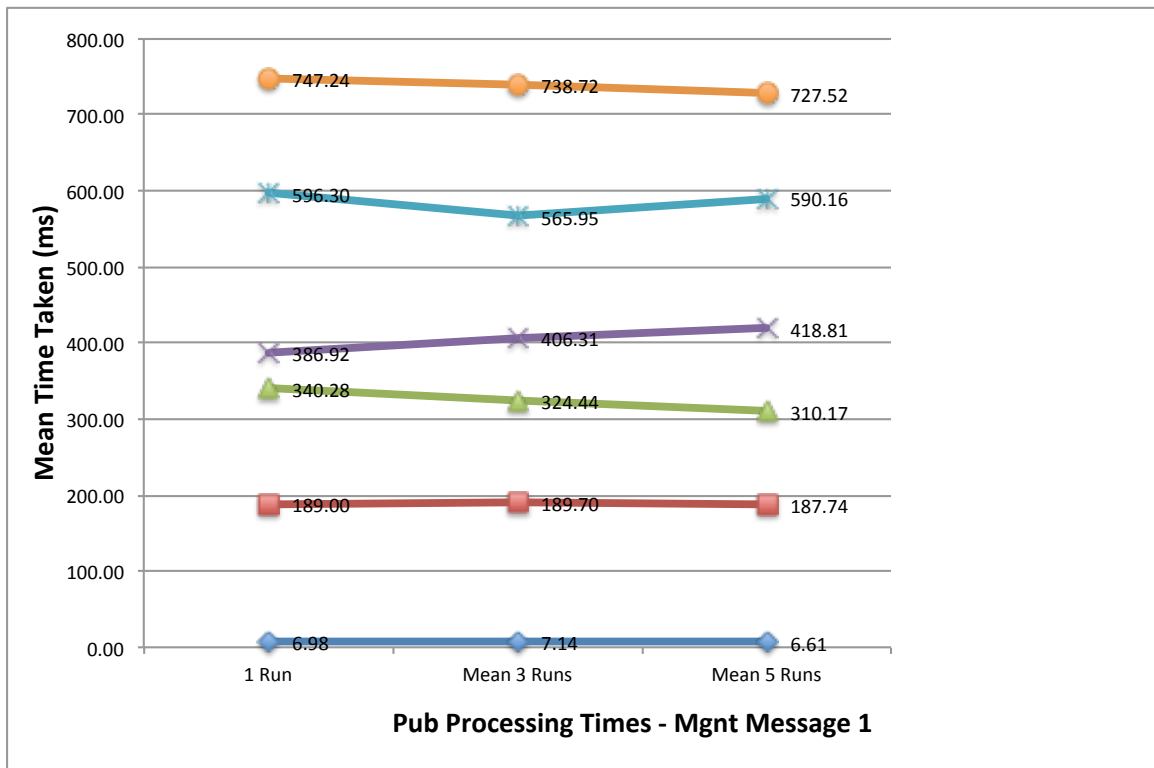
Subscription Tree Search Times



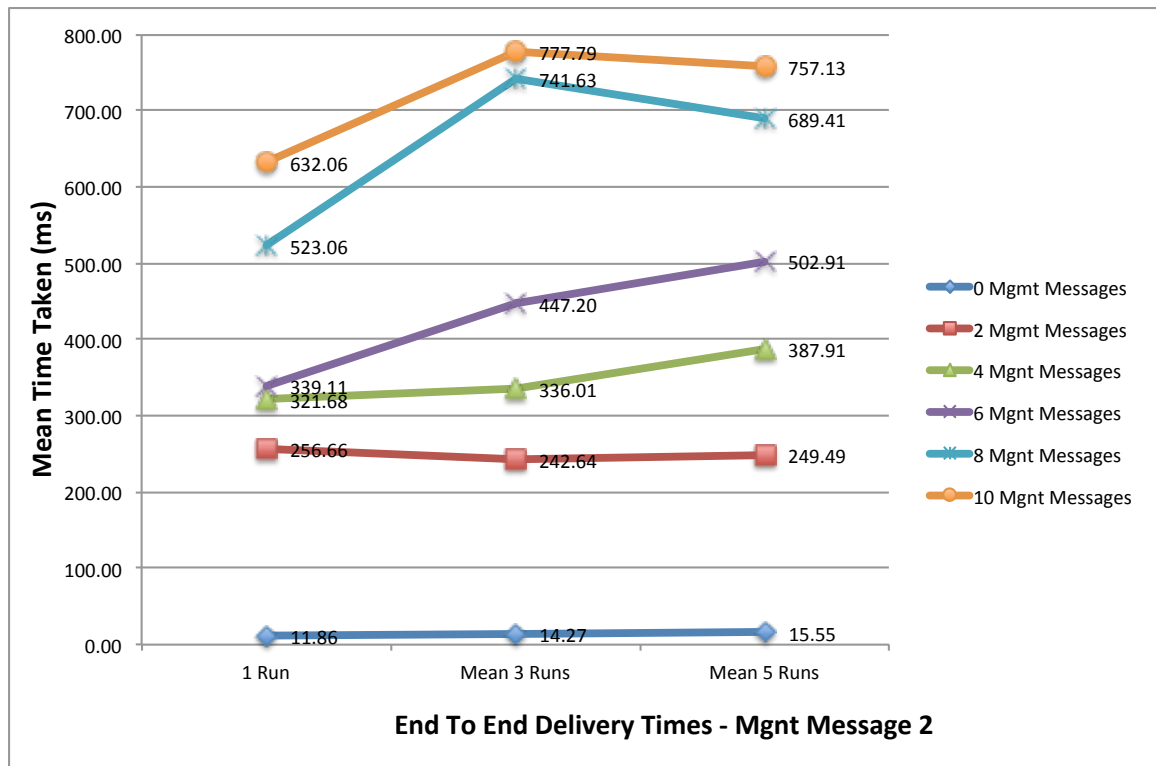
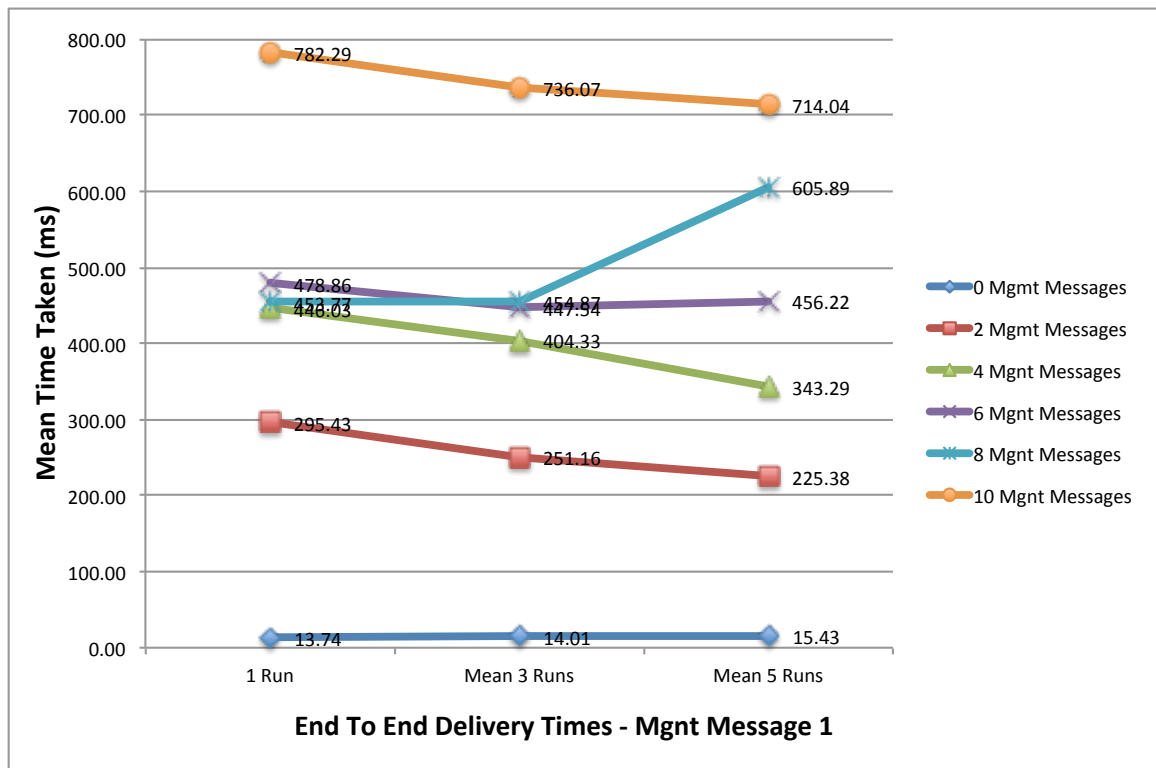
Subscription Processing Times



Publication Processing Times



Pub-to-Sub (End-to-End) Delivery Times



Moving Broker and Moving All Subscribers

