A DECENTRALIZED CONTENT-BASED COMMUNICATION FRAMEWORK FOR SUPPORTING DECOUPLED GRID INTERACTIONS

BY ANDRES QUIROZ HERNANDEZ

A Thesis submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Master of Science
Graduate Program in Electrical and Computer Engineering
Written under the direction of
Professor Manish Parashar
and approved by

New Brunswick, New Jersey
May, 2007

ABSTRACT OF THE THESIS

A Decentralized Content-based Communication Framework for Supporting Decoupled Grid Interactions

by ANDRES QUIROZ HERNANDEZ

Thesis Director: Professor Manish Parashar

This work presents a decentralized and content-based communication framework based on Web Services Notification (WSN) to enable monitoring and loosely-coupled interactions in distributed and particularly Grid systems. Web services have emerged as one of the key enabling technologies for Grid systems, providing platform-independent interactions between distributed applications and resources. The WSN specification is a set of web service standards that define protocols for realizing the publish/subscribe communication pattern. Existing implementations of WSN fail to address key issues particular to large-scale Grid systems, such as the dynamic participation of distributed computing nodes, the heterogeneous services provided by nodes across physical and organizational domains, and the need for efficient messaging mechanisms.

The specific contributions of this work are as follows: 1) Design and development of a notification service implementing the WSN specifications based on a decentralized content-based addressing and messaging infrastructure; 2) Design and evaluation of self-optimization mechanisms for message exchanges within the notification system; 3) Design and analysis of a novel two-level overlay structure that extends the messaging infrastructure for efficiently interconnecting separate individual physical or organizational groups of computation nodes without resorting to designated nodes that can become bottlenecks or single points of failure; 4) Development of a mechanism for self-monitoring of peer-to-peer systems that is able to detect anomalies and/or trends in system behavior as a direct application of the notification service.

The experimental evaluations presented demonstrate the performance and scalability of the infrastructure as well as the effectiveness of the self-monitoring mechanism.

Acknowledgements

I would like to express my special gratitude toward my advisor, Dr. Manish Parashar, who gave me this opportunity and guided me in the completion of this work. To Dr. Ivan Marsic, who also played a special role in getting me here, and to Dr. Deborah Silver for her participation in my thesis committee. And, of course, I am always grateful to my family and friends, here and back home, whose support is the key ingredient to all of my accomplishments.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The goal of the Grid infrastructure is to combine intellectual and physical resources spanning multiple organizations and disciplines, and provide vastly more effective solutions to scientific, engineering, business and government problems [37]. The realization of this goal requires the meaningful exchange of information that enables interactions and coordination between these distributed resources. We observe that there are different degrees of coupling among Grid resources at which this information exchange takes place, made evident by the concept of virtual organizations, discussed in [17]. Virtual organizations are organizations that arise opportunistically for a particular purpose and that combine resources provided by multiple concrete organizations. Thus, interactions within virtual organizations are loosely-coupled and driven by the events that must be exchanged to coordinate the execution of tightly-coupled resources that perform clearly defined tasks within concrete organization domains.

As a simple example, consider a corporate environment in which interactions may be defined between the sales and production departments, and with providers to coordinate delivery and production schedules. Each department defines tightly-coupled workflows for their particular operation. Depending on different situations or events, these workflows may be invoked and exchange information. An unexpected event or error may trigger the execution of contingency workflows that may involve other departments. Figure 1.1 illustrates this example.

Because of the close and frequent interactions of the tasks that make them up, resources that are more tightly-coupled can use application-specific communication protocols and interfaces. Loosely-coupled interactions across organizational domains, on the other hand, require standard communication protocols and interfaces. Web services have emerged as one of the key enabling technologies for Grid systems, providing

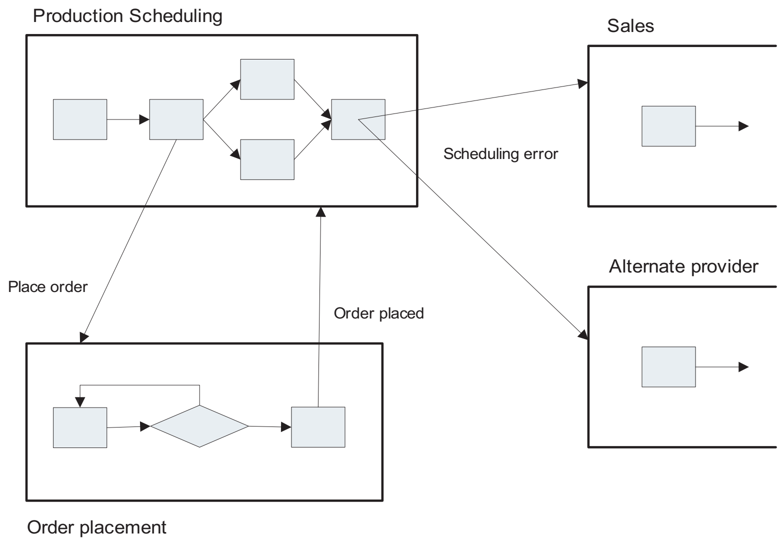

Figure 1.1: Example of two-levels of workflow coupling in a production environment

platform-independent interactions between distributed applications and resources. The WS-Notification (WSN) specification [35] is a set of web service standards that define protocols for realizing the publish/subscribe communication pattern. If implemented to address the key issues of dynamism and scale in Grid systems, an implementation of the WSN standard is a suitable solution for realizing loosely-coupled, event-driven communication in Grids.

1.2 Problem Description

Most existing implementations of the emerging WSN standard are essentially bindings to programming languages that include extensible API's for application developers, and are thus not meant to address system specific issues. This work addresses the problem of designing a suitable implementation of the WSN standard as a platform for efficient loosely-coupled communication in Grid systems. This design must thus deal with issues and challenges specific to Grid environments, leveraging available technologies if possible. Below, we describe the key design aspects as they relate to these issues.

- *Dynamism and Scale*: Dynamism in Grid systems is evident in 1) the availability of resources (referred to henceforth as nodes), as some nodes are made available and other leave the system at runtime, and 2) the alternative provision of the same sets of tasks by different nodes. This makes it infeasible for nodes that must execute and interact in the execution of different tasks to keep track of the names or addresses of the all the other nodes with which this interaction may take place. The decoupling of interactions provided by the publish/subscribe communication pattern is meant to deal with this issue, but the potentially many interacting entities and types of interactions mean that the communication system must be distributed and decentralized to provide a scalable service. It also means that the addressing scheme used to establish interactions must be flexible and expressive enough to encompass and distinguish this potential number of interactions without being fixed like a name or absolute address, but it must conform as closely as possible to the notification standard.

- *Infrastructure Support for Inter and Intra-domain Communication*: As we have previously discussed, tasks, and the nodes that execute them, tend to be grouped logically or physically within different interacting virtual or concrete organizations. Within these groups, nodes will tend to interact more closely and frequently than they do between them. It is thus desirable for the underlying infrastructure that interconnects these nodes to reflect and take advantage of this grouping to optimize communication. Grid applications are often built on top of an abstraction of a physical network called a network overlay, which basically provides an addressing scheme and routing mechanisms for the overlay nodes. Given this grouping of nodes, it is not very reasonable to assume that a communication platform for a large dynamic Grid will be built on a single overlay structure. Instead of a single overlay that does not incorporate group knowledge and can thus be inefficient as messages arbitrarily traverse group boundaries, a Grid system can be composed of several interconnected overlays. This structure may be seen as an overlay of overlays or, similarly, as a hierarchical overlay. Although some solutions have been proposed for the design of hierarchical overlays, most of them depend on specialized nodes that can become bottlenecks or single points of failure.

Finally, we consider the importance of the problem of monitoring in distributed systems, including peer-to-peer and Grid systems, which consists of detecting anomalies and/or failures of system components, so that corrective measures can be applied in a timely and effective manner. Recognizing that a solution to this problem requires an effective and efficient notification infrastructure, and that most components in these systems possess the communication and computation capability to share and process data online, we explore the problem of in-network, decentralized data analysis for self-monitoring as a direct application of notification and as a useful service that can be leveraged across Grid applications. Self-monitoring, especially of distributed systems, is challenging because all possible problems or causes of failure are not known beforehand and thus cannot be directly encoded into an automated monitoring mechanism. Existing self-monitoring techniques rely on the centralized processing of monitoring data to produce models that can then be deployed for automated application in real time. The derivation of these models often requires sophisticated processing and data mining techniques. However, for systems such as distributed peer-to-peer networks, the amount of monitoring data that must be analyzed may make the application of centralized techniques infeasible.

1.2.1 Objectives

In order to achieve the main goal of developing an efficient distributed platform to provide a scalable notification and monitoring service for Grid systems, this work addresses the following specific objectives:

- Design a content-based addressing scheme that can be adapted for an implementation of the WS-Notification specification.
- Develop an efficient distributed notification service that implements the WS-Notification interfaces using the content-based addressing scheme and leveraging a content-based communication platform.
- Design and evaluate mechanisms for optimizing message flows generated by the notification service.

- Devise and test a hierarchical overlay structure that allows logical and/or physical grouping of nodes and that does not rely on designated nodes for inter-group communication to avoid bottlenecks and single points of failure.
- Build a decentralized solution for self-monitoring on top of the notification infrastructure.

1.3 Approach and Contributions

For our distributed and decentralized implementation of the WS-Notification standard, we leverage a messaging framework called Meteor [28], which is based on a rendezvous messaging model and content-based addressing scheme. In order to more closely reflect the physical and logical grouping of interacting nodes, we adapt the framework with a hierarchical overlay design. We then apply our notification infrastructure to provide a solution for self-monitoring of peer-to-peer systems. Specifically, the following contributions result from this approach:

- Design and development of a distributed content-based WS-Notification service based on the the Meteor rendezvous-based messaging platform.
- Incorporation of optimization mechanisms for message exchanges within the notification system.
- Design and evaluation of a two-level structured overlay that does not rely on designated nodes for communication between groups of nodes, and that uses a novel mechanism to obtain efficiency in inter-group routing.
- Adaptation of the Meteor platform to the two-level overlay.
- Development of a mechanism for distributed system self-monitoring and a decentralized clustering analysis engine based on the distributed content-based notification platform.

Figure 1.2 shows our basic system architecture. The white boxes show particular contributions of this work.

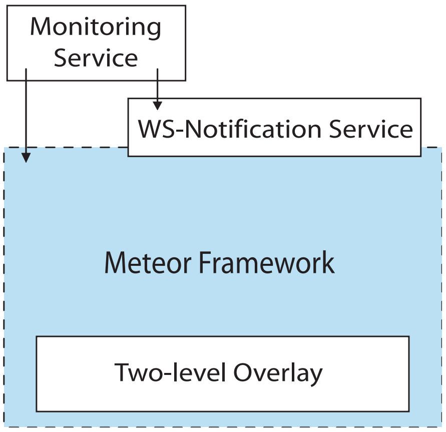

Figure 1.2: Architectural overview of the system

1.4 Thesis Overview

This thesis is organized as a bottom-up description of the components of the notification insfrastructure. Chapter 2 examines the background and related work in distributed publish/subscribe systems and WS-Notification. It also covers the details of the content-based communication infrastructure around which this work is developed.

Chapter 3 describes the two-level structured peer-to-peer overlay, which divides nodes into communicating groups of local peers. Inter-group connectivity is maintained without resorting to the use of group heads. The chapter further analyzes the benefits of such a structure, and shows and evaluates its efficient routing and search properties.

The design of the notification service, centered around the Web Services Notification standards, is then described in Chapter 4. This design is based on a content-based addressing scheme that extends the topic-based addressing scheme described by the WS-Notification specification, and the definition of notification operations in terms of rendezvous-based communication. The messaging optimizations meant to reduce the flows of messages within the system are also described and evaluated.

Chapter 5 describes a mechanism for self-monitoring of peer-to-peer systems that directly takes advantage of the content-based notification infrastructure. Besides using the notification service to enable selective subscriptions to situations of interest in a monitored system, the self-monitoring mechanism leverages the clustering properties of the content-based routing mechanism of the communication infrastructure for supporting a novel in-network, decentralized data analysis scheme.

Finally, Chapter 6 presents our conclusions and considerations for future work. Implementation details are included as Appendix A.

Chapter 2

Background and Related Work

In this chapter, we review key concepts for publish/subscribe, as well as existing distributed publish/subscribe systems. We focus on a specific subset of these systems, those which are based on Distributed Hashtables (DHTs), because of their decentralized and self-organizing nature. We then give a brief overview of the WSN standard and its existing implementations, and also give an architectural and functional description of the Meteor framework, as it is the principal enabling technology used in this work. Other specific related work is covered in subsequent chapters.

2.1 Publish/Subscribe

2.1.1 Types of Publish/Subscribe

The publish/subscribe paradigm enables many-to-many communication between senders and receivers without requiring these entities to explicitly know each other's identity and/or contact information. In other words, it decouples the interaction between senders and receivers by introducing common abstract identifiers to which senders publish and from which receivers obtain data. The shared knowledge of the identifiers includes the type of data that is expected to be exchanged over them. Eugster [16] distinguishes three types of publish/subscribe patterns or addressing schemes depending on the type and level of the common knowledge that entities use to interact. These are:

- *Channel-based*: In channel-based publish/subscribe, a (finite) number of channels or addresses identify different groups of communicating entities. Usually, this type of publish/subscribe system corresponds to network-level technologies, such as IP Multicast, and can be assigned, either statically or dynamically, to applications or data flows. However, there is no defined semantic association between a channel and the data transmitted over it.

- *Topic-based*: Topic-based publish/subscribe is similar to its channel-based counterpart, but is considered at a higher (application) level abstraction. Here, each one of a fixed number of syntactic tags (topics) identifies a certain type of data, according to their semantic interpretation. Although topic spaces can be hierarchical, so that subscriptions can be done at different resolutions to control the amount and specificity of data, all topics must be defined *a-priori*.

- *Content-based*: Content-based publish/subscribe is also an application-level abstraction in which the shared knowledge between senders and receivers consists of a fixed number of semantic criteria that are used to determine the distribution of data. In theory, these criteria take the form of boolean functions $f_i = \Sigma^* \rightarrow \{0, 1\}$ that take a message as input and output 1 if the message matches criteria i and 0 otherwise. The fact that messages themselves (their content) are used to classify data and determine subscription groups is the reason the pattern is called content-based. The advantage of this pattern is that while the amount of shared knowledge that must be known *a-priori* (the evaluating criteria) is relatively small, the possible classifications for data can be as varied and specific as messages themselves. However, since arbitrary functions f_i may be too specific, complex, or costly to implement, content-based systems often define a set of semantic attributes and their value ranges that can be combined to classify and identify data. While not as flexible as generic criteria, these attribute/value pairs still constitute a relatively small amount of shared knowledge and can be combined to produce a geometrically large set of identifiers for message flows.

2.1.2 DHT-based Distributed Publish/Subscribe Systems

Foundations: Overlays and DHTs

A network overlay is an abstraction of a physical network that provides an identifier or addressing scheme for a subset of network nodes, maintains a set of logical links between them, and provides routing mechanisms to traverse these links to enable node interaction. Overlays are essential for distributed systems because the addressing abstraction that they provide decouples participating nodes from actual physical addresses, so that different entities may fulfill a single role at different times without forcing other entities to explicitly keep track of addressing changes that occur because of dynamic node participation.

Network overlays can be either structured or unstructured, depending on whether or not they maitain a strict topology (i.e. fixed properties for the number and layout of the physical links between nodes). Unstructured systems [33] do not rely on any specific topology, and thus use flooding techniques to process requests. Their strength is that they can support complex content-based searches, since requests can be evaluated against the content at each node as the search advances. However, the high overhead of flooding forces these systems to limit the scope of the searches, and as a result, they cannot guarantee that all corresponding matches for a search will be found. Since most publish/subscribe systems (at least those intended for scientific or commercial applications) require better search guarantees, unstructured overlays are not generally used as a basis for them. While a number of techniques have been proposed to reduce the overhead of searches in unstructured networks [13, 33] in order to produce better search results, the fundamental lack of search guarantees remains for a reasonable sized system.

Publish/subscribe systems are usually built using some sort of structured overlay network because they provide search guarantees, bounds on the number of hops for message delivery within the network, and some degree of self-management and fault tolerance with respect to the addition/removal of nodes. Though they are not as scalable as unstructured overlays due to the overhead that results from maintaining the required structure, the most well-known designs scale reasonably well, especially, as has been mentioned, with respect to search. Structured overlays such as Chord [47], Pastry [40], and CAN [39] are often used as the basis for the Distributed Hashtables (DHTs) used to implement publish/subscribe systems.

A DHT is a system that associates objects identified with a set of keys with nodes in a distributed system and that provides the mechanisms to store or retrieve these objects transparently through a put/get interface. DHTs are generally used as the backbone of a distributed publish/subscribe system, as they provide the means for storing and retrieving subscriptions. DHTs implemented directly using the structured overlays mentioned above typically use consistent hashing functions to assign node identifiers to unique content identifiers. This means that they are guaranteed to find all subscriptions that exactly match a given query, but they do not support the complex queries necessary for content-based search.

Content-based Publish/Subscribe

With this foundation, designing content-based publish/subscribe systems requires an efficient mapping between complex content descriptors and node identifiers in the overlay network, as well as efficient techniques for routing and matching based on these content descriptors. This is formalized in [6] as it applies to the problem of content-based publish/subscribe. In this work, the authors discuss a generic model of mappings from the content-based keyword space to the overlay's index space and from this index space to overlay nodes. They distinguish between the mappings used for subscriptions and events, and establish that the necessary condition that these mappings must meet is the "mapping intersection rule", which states that the intersection of the sets of nodes returned by a subscription mapping and an event mapping of an event satisfying that subscription must be non-empty. We will discuss some approaches to content-based publish/subscribe that basically differ in the type of mappings used.

In [48], a method is proposed for building topic strings from the content descriptors of subscriptions and event messages by concatenating attributes and values based on predefined classifications of keys called *schema*. These topics are then used as keys for any DHT (the Scribe system [9] built on Pastry is used). Attribute ranges in subscriptions are handled by predefining range intervals that are added to the domain of attribute values. The definition of the *schema*, which further limits the flexibility of content-based indexing by constraining the combinations of attributes used for indexing, and which requires fine-tuning of the range intervals for load balancing, limits the practicality of this approach.

Aekaterinidis and Triantafillou also proposed a DHT-based system that is independent of DHT design and implementation and that can handle subscriptions based on partial strings [1]. Their approach is based on decomposing subscriptions per attribute according to whether they have equality, prefix, or suffix constraints. For each attribute, the available value is then mapped to a DHT node and stored in separate lists according to this classification. Event matching is done by mapping all possible prefixes and suffixes of attribute values and finding matches in the corresponding lists. This could be improved if the mapping of subscriptions preserved the locality of attribute values (for example, the mapping of “comp*” and “comput*” will have completely unrelated values, so that the mapping has to be applied to all prefixes). Also, since it is designed for string matching, this approach cannot handle numeric attributes.

The Meghdoot system [23] and Meteor use DHT-dependent mappings that do preserve the locality of content descriptors, thus reducing the overhead of search. They also support partial string as well as numeric range queries. Meghdoot directly builds a multidimensional key space for CAN from an adaptation of the multidimensional space obtained from the attributes, so that locality in both is the same. Meteor relies on a one-dimensional overlay, namely Chord, and thus relies on a more complex mapping to preserve locality. Because Chord generally outperforms CAN in terms of the number of nodes involved in routing ($O(\log n)$ vs $O(dn^{\frac{1}{d}})$, where d in this case is twice the number of attributes) and its one-dimensional topology is less costly to maintain, Meteor is considered better suited to support the present work. Further details of the Meteor framework are included below in Section 2.3.

To our knowledge, none of these systems are as yet used to implement the WSN standards, which are briefly described in the next section.

2.2 Web Services Notification

The WSN specification consists of 3 interrelated standards: WS-BaseNotification (WSBN) [21], WS-BrokeredNotification (WSBrN) [11], and WS-Topics (WST) [49]. WSBN specifies the basic elements of the notification pattern: the NotificationConsumer (NC) that accepts notification messages, the NotificationProducer (NP) that formats and generates these messages, and the Subscription, a consumer-side initiated relation between a producer and a consumer. The only fixed field in a subscription is the consumer reference, which by itself indicates the consumer's interest in all of the notifications generated by the producer to which the subscription was made. Optionally, a subscription can contain a filter, specified as a FilterType element, that forces a producer to send only those notifications that pass (match) the filter. WSBN does not regulate the syntax or use of the FilterType element, but suggests three basic types: topic filters, message content filters, and producer properties filters. WSBN also regulates subscription management, which the consumer can perform given the reference it receives in response to a subscription. This reference is meant to contain enough information to enable it to contact and interact directly with the subscription as a resource, as defined by WSRF. Thus, subscription operations (unsubscribe, renew, pause, and resume) do not include a subscription reference as a parameter. In addition to the push-style pattern of notification, where producers send notifications directly to consumers, WSBN defines a pull-style pattern, where messages are stored at a pre-defined location (a pull-point) until they are retrieved by the consumer.

WSBrN defines the NotificationBroker (NB) entity and its expected functionality. A notification broker is an intermediary Web service that decouples NC's from NP's [11]. A broker is capable of subscribing to notifications on behalf of consumers and is capable of disseminating notifications on behalf of producers. Consumers and producers thus interact dynamically and anonymously through the NB without the need for explicit knowledge of each other's identities or locations. Management of this knowledge is delegated to the broker. A NB essentially implements the NC, NP, and other interfaces defined in WSBN. As a specific functionality, a notification broker can accept producer registrations, in order to realize the demand-based publishing pattern. Using this pattern, publishers avoid the (possibly expensive) task of creating notifications when no subscriptions (and, thus, no consumers) exist for them. To this end, a NP must register with the NB, providing a set of topics. When subscriptions are made that correspond to or include topics in a particular producer's registration, the NB subscribes to the

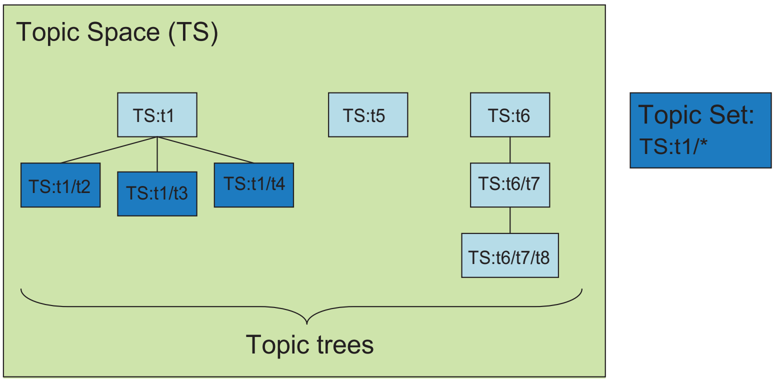

Figure 2.1: Example topic space and topic expression. The darkened topics are those represented by the expression on the right.

producer for those topics. Only then does the producer start sending notifications.

Finally, WST tries to standardize the way in which topics are defined, related, and expressed. It defines the notion of a topic space, where all of the topics for an application domain should be defined and organized, possibly in a hierarchical way. A topic expression is the representation of a particular topic or range of topics. The syntax of a topic expression is identified by the topic expression's dialect. WST defines three dialects: Simple, Concrete, and Full. A simple topic expression is just the qualified topic name. A concrete topic expression is used for hierarchical topic spaces, and is given in a path notation, such as in `myNamespace:news/tv/cnn`. Here `myNamespace` identifies the topic space, and each of the subsequent identifiers belong to successively deeper levels in the hierarchy. A full topic expression is the same as a concrete expression, except that it uses special operators and wildcard sequences for spanning multiple topics within the topic hierarchy. Figure 2.1 shows an example of a hierarchical topic space and a full topic expression that spans a group of topics.

2.2.1 WSN Implementations

To date, there are several implementations of WSN, including Apache's Pubscribe [4], for Java, WSRF.NET from the University of Virginias Grid Computing Group [50], for Microsoft's development platform, pyGridWare [38], a Python-based implementation, and GT4 from the Globus Toolkit [22], with bindings for both Java and C.

Apache's project is derived from GT4-Java. The primary focus of these implementations is WSRF, and, as a result, they provide different levels of functionality for WSN. For example, the pyGridWare and GT4 implementations of WSN are meant primarily for providing notifications about the state of resource properties. Pubscribe extends these capabilities and fully supports both WS-BaseNotification and WS-Topics, but does not implement WS-BrokeredNotification. WSRF.NET, which was developed using ASP.NET and the IIS infrastructure, supports all of the specifications. A thorough comparison of these implementations can be found in [25].

The above implementations are meant to be development tools, providing technology-specific bindings of the standards and extensible API's. Like the standards themselves, they do not address the issues that arise when actually composing systems that make use of the notification protocols and standards, such as service discovery, and efficient and scalable routing of requests and messages. These issues have been addressed in the context of messaging infrastructures such as the Enterprise System Bus (ESB) architecture [34] [45], which mediates the interactions of different web services, including WSN service implementations, by service virtualization. An ESB hides implementation and location details of the services that register to it and is capable of spanning wide area networks and involving multiple infrastructure servers. However, as its name suggests, an ESB is an enterprise-level solution and there is no reference implementation for it. NaradaBrokering [36] is a distributed middleware framework that supports peer-to-peer systems and message-oriented interactions. It implements a variety of messaging technologies and is currently working on an implementation of WSN. The objectives and approach of the NaradaBrokering framework are essentially the same as those that underlie the present work; it manages a network of brokers through which end systems can interact, providing scalability, location independence, and efficient routing. The difference is that Narada brokers are organized in a structure which must be maintained through tighter coupling and control mechanisms that do not allow uncontrolled connections and disconnections.

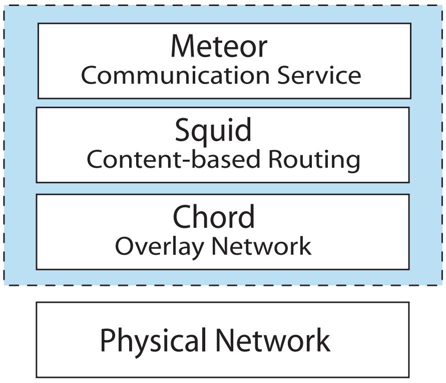

Figure 2.2: The Meteor framework

2.3 The Meteor Framework

Meteor [28] is a content-based communication platform for peer-to-peer systems based on a rendezvous messaging model [46]. The Meteor framework is composed of the Meteor service itself and a content-based routing infrastructure built on a structured peer-to-peer overlay. Figure 2.2 shows the framework architecture and each of the component technologies, described below.

2.3.1 Squid: Content-based Routing Engine

Squid [44] implements a dynamic mapping between a content-based identifier space and a dynamic set of peer nodes. It is built as a distributed system on top of the indexed peers, which are organized using the Chord [47] overlay network. Its key innovation is a dimension reducing indexing scheme and routing mechanism that effectively maps descriptors in a content-based information space to physical peers in the overlay network. Squid dynamically divides the identifier space among the peer nodes, so that disjoint continuous subspaces of the identifier space are assigned to each participating peer node. Squid guarantees that all existing peers to which a given identifier corresponds can be reached with bounded costs in terms of the number of messages and the number of nodes involved.

As stated, the key of Squid's functionality is the indexing scheme that enables it to preserve locality while mapping the content-based data elements to the overlay's one-dimensional index space, so that elements that are close in the information space will

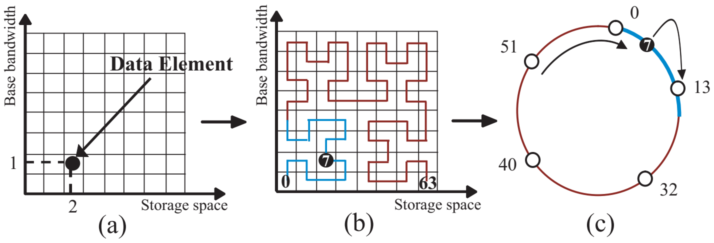

Figure 2.3: Mapping from a multidimensional keyword space to a node in the overlay network

be close in the index space. This enables Squid to efficiently handle complex content-based identifiers with partial keywords, wildcards, and ranges, as these queries map to a reduced number of peer nodes.

The mapping of keywords to the index space, shown in Figure 2.3 is accomplished by using a locality preserving mapping called a Hilbert Space Filling Curve (SFC) [41]. An SFC is a continuous, recursively generated mapping from a d-dimensional space to a one-dimensional space. In the figure, the Hilbert SFC traverses a two-dimensional information space used to index some computation resource with regard to bandwidth and storage (Figure 2.3 (a)). The values of these attributes for a particular resource correspond to a coordinate in this two-dimensional space, and, consequently, to a point on the SFC (Figure 2.3 (b)). The dimension of the SFC maps to Chord's m-bit identifier space, so that this point corresponds to a Chord identifier (Figure 2.3 (c)).

Nodes in the Chord overlay have identifiers in this same identifier space, assigned to them randomly upon joining, and each node is responsible for the identifiers between its predecessor (the node with the immediately preceding id) and itself. Chord provides a lookup operation by which the node responsible for a given identifier (its successor) can be determined. Beside predecessor and successor links, Chord nodes maintain strategically chosen references, in a structure called a finger table, of other nodes in the ring, so that lookup operations can be resolved in $O(\log n)$ number of messages (hops), where n is the number of nodes in the system.

Notice that, because of the locality preserving quality of the SFC, data elements that are close (in this case, lexicographically close) in the multidimensional index space will likely be mapped to indices that are local in the one-dimensional index space. Because

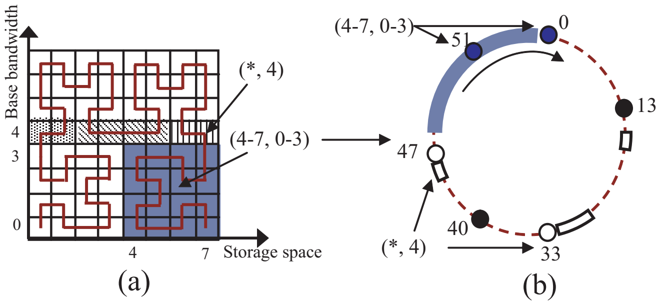

Figure 2.4: Searching the system: (a) regions in a 2-dimensional space defined by the queries (*, 4) and (4-7, 0-3); (b) the clusters defined by query (*, 4) are stored at nodes 33 and 47, and the cluster defined by the range query (4-7, 0-3) is stored at nodes 51 and 0.

of this, close indices will likely be mapped to nodes that are close in the overlay, or even to the same node. Figure 2.4 illustrates the search process using complex queries (e.g. ranges and wildcards). As Figure 2.4 (a) shows, these queries are translated into a collection of segments (called clusters) in the one-dimensional space. The appropriate nodes are contacted using the lookup mechanism provided by the overlay.

The Squid decentralized routing engine is optimized to distribute the identifier resolution at multiple nodes in the system, ideally the nodes to which the identifier corresponds, minimizing the communication and computational costs. The optimization takes advantage of the recursive nature of the SFC-based mapping scheme and generates the minimum number of index clusters for each query, such that all the corresponding nodes are identified. Details about this optimization can be found in [44].

2.3.2 Meteor Associative Rendezvous Communication Service

The Meteor communication service is a kind of DHT based on associative rendezvous, a paradigm for content-based decoupled interactions [28]. The peers that implement the service function act as rendezvous points (RPs), which, like in typical rendezvous-based communication [16], serve as meeting places that link communicating nodes. Associative rendezvous simply means that the choice of particular RPs by a node at some point in time, and thus the set of nodes with which that node will interact at that time, is transparently and dynamically determined based on its interests, expressed as a content-based profile. In other words, nodes are dynamically associated with each other via RPs based on common profiles.

Instead of a DHT put/get interface, Meteor exposes a single symmetric **post** primitive, which accepts message triples of the form (*header*, *action*, *data*). An entity that wishes to communicate through Meteor will invoke the **post** primitive on a Meteor peer, specifying in the *header* its profile as a set of attribute values, ranges, or wildcards, an appropriate *action* defined as a Meteor reactive behavior (see below), and a *data* payload. The *header* also contains credentials and other information. A Meteor peer runs on top of a Squid peer, and, correspondingly, is responsible for a portion of the content-based identifier space from which profiles are drawn. The Meteor peer uses Squid's routing engine to deliver a message to corresponding RPs based on the profile included in the message header. Squid will ensure that the message is delivered to the RP(s) responsible for the given profile, as described in the previous section.

Meteor's reactive behaviors define the way that a RP handles the messages that it receives via the routing mechanism, differentiating the results obtained with different calls to the **post** primitive. The Meteor service has the following predefined behaviors, the functionality of which is illustrated in Figure 2.5. Notice that the predefined behaviors implement a pull-style system, in which receivers explicitly retrieve messages from the rendezvous point, possibly after being notified of its existence. Extensions of the Meteor service may redefine these behaviors or define new behaviors to implement other messaging patterns, such as a push-style publish/subscribe.

- `store`: The RP stores the profile and executes matching profiles that contain a `notify_data` action.
- `retrieve`: The RP matches the received profile with existing profiles with `store` action and sends the data associated with the matched profile(s) to the requester.
- `notify_data`: The RP stores the received profile and executes matching profiles that contain a `notify_interest` action.
- `notify_interest`: The RP stores the profile, matches this profile with existing profiles with `notify_data` action, and sends a message to the requester if any such matches exist.

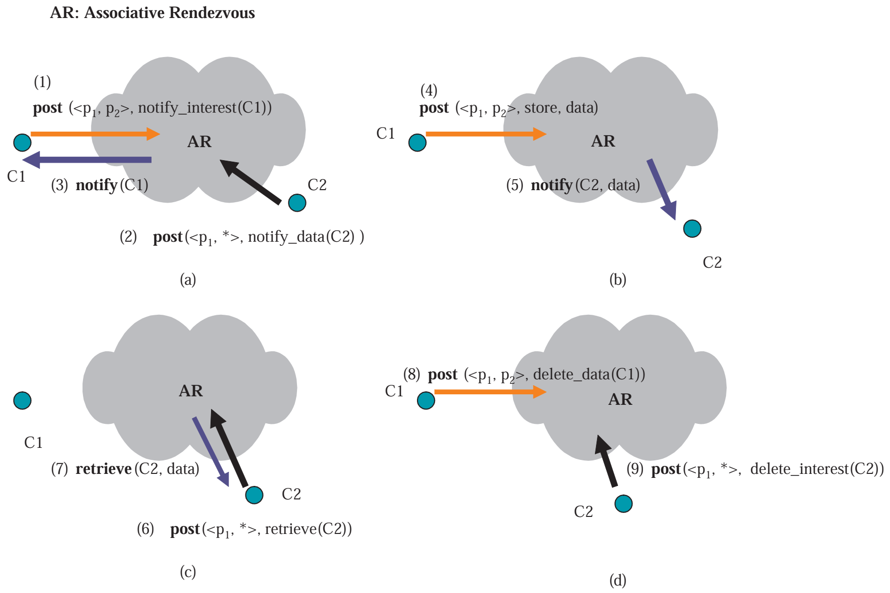
AR: Associative Rendezvous
(1)
post (<p1, p2>, notify_interest(C1))
AR
C1
(3) notify (C1)
C2
(2) post (<p1, *>, notify_data(C2))
(a)
(4)
post (<p1, p2>, store, data)
C1
AR
(5) notify (C2, data)
C2
(b)
AR
C1
(7) retrieve (C2, data)
C2
(6) post (<p1, *>, retrieve(C2))
(c)
(8) post (<p1, p2>, delete_data(C1))
C1
AR
(9) post (<p1, *>, delete_interest(C2))
C2
(d)

Figure 2.5: An example illustrating Meteor's associative rendezvous and reactive behaviors

- `delete_data`: The RP deletes matching profiles that contain the `store` action.

- `delete_interest`: The RP deletes matching interest profiles.

Chapter 3

Two-level Overlay

3.1 Analysis and Justification

We saw in Section 2.1.2 how structured overlays generally provide the basis of DHT implementations and distributed publish/subscribe systems. An important problem of many structured overlay designs in terms of performance and scalability is that they do not take locality into account. This means that the neighbor relationship upon which overlay links are based does not necessarily correspond to the actual physical links connecting the neighbor nodes. While in unstructured overlays this problem tends to be minimized because of the ad-hoc way in which nodes join the network, usually through searches for local peers, in structured overlays nodes are generally assigned a place in the topology randomly, by hashing, or by some balance-preserving function.

Physical locality is not the only type of locality that can be considered. In effect, the concept of virtual organizations that underlies Grid systems denotes logical groups that are formed for a particular purpose or under a particular administrative, organizational, or application domain. Such locality should be reflected in the topology of overlays used for Grid systems because most communication will tend to take place between nodes in these local groups, so that exchanging these messages through nodes outside of these groups is wasteful or even contrary to policy. However, the existence of these groups should not preclude communication between nodes in different groups, and furthermore, this communication should still be made efficient and, in the case of structured overlays, provide similar guarantees as regular overlay deployments.

A straightforward way to build an overlay structure that incorporates locality is by building an overlay for each local group and interconnecting these overlays to form a higher level structure, or overlay of overlays. This structure, which is essentially hierarchical, fits nicely into the context of virtual organizations in Grids, where each organization can build and maintain its own overlay and communicate via inter-overlay links.

We have devised a method for constructing a two-level structured peer-to-peer overlay, which has advantages over existing designs under specific assumptions. The designed overlay is composed of groups of local peers, within which most interactions, including routing, indexing, and data placement, take place. Inter-group connectivity is maintained without resorting to the use of specialized nodes, referred to as group heads. The key idea in this overlay design is the use of a uniform identifier space for all groups that allows any node to find its place and perform operations in the remote group with an operation called a *virtual join*. Nodes that perform virtual joins in remote groups learn but do not modify the overlay structure of the remote group to exploit search locality in remote operations, thus improving their efficiency.

Before explaining the details of the overlay design, the next sections provide some references of related work, as well as further theoretical and practical motivation for a hierarchical, and particularly two-level overlay.

3.1.1 Hierarchical Overlays

Typically, designs for hierarchical overlays use the concept of group head or superpeer, so that one node or subset of nodes in each group are used to form the higher level overlay(s) that connect the lower level groups [20, 43]. Depending on the design of higher level overlays and routing protocols, these nodes can become bottlenecks or single points of failure. However, different techniques, such as voting, rotation, replication, etc. can be used to minimize this potential for group head(s).

Several structures do maintain hierarchies without group heads. In [10], hierarchy is introduced by using hierarchical identifiers for the overlay nodes. Routing in this design is prefix-based, but it requires specialized root nodes as in DNS routing, which in a way are like group heads. The structure described in [15] also uses specialized identifiers that preserve locality when ordered. Efficient routing is performed in a similar way to Chord by maintaining a list of references, known as skip lists, to progressively more distant nodes. Ganesan et al. [19] describe Canon, in which links are constructed between nodes in different groups as if constructing a single layer cluster, but keeping intra-cluster and inter-cluster links separate and limiting the number of inter-cluster links to bound the average number of links per node. The Cyclone overlay [42] is similar, but claims to provide more flexible routing through multiple paths. The main problem with these overlays is that they are designed and evolve as a whole, and all changes propagate throughout the structure much as they would in a flat overlay. The context considered here rather favors designs in which individual structures can evolve and then be combined opportunistically to form a single hierarchical overlay.

3.1.2 Analysis of Communication Costs

When hierarchy is introduced into an overlay design by grouping nodes at a low level according to some locality metric, the links between these nodes can be implicitly differentiated as local (those between nodes in the same group) and remote (those between nodes in different groups). It can be assumed, according to the metric of locality used for grouping, that the cost of using local links is less than that of using remote links. The following analysis shows how a two-level overlay has better average search times than a flat overlay, where a search is defined as the act of routing to a node that can respond to (resolve) a particular query.

Let the average communication cost between two nodes in the same group be t, and that between two nodes in different groups be T, such that $t << T$. Let $f(x)$ be the average number of overlay hops to resolve a query in an overlay with x nodes. Assuming n nodes per group and k groups ($N = nk$ total nodes) and that all groups are connected, the average search costs in a one level and two level overlay are obtained as follows, given that the query can be resolved in only one group:

One-level overlay

The probability that a particular hop is to a node on the same group is

$$\frac{n-1}{N-1} \approx \frac{1}{k}$$

Based on this probability, the average cost per hop (h) is the sum of the cost for each kind of hop (local or remote), weighted by the probability of that hop. This is:

$$h = \left(\frac{1}{k}\right) t + \left(1 - \frac{1}{k}\right) T$$

Thus, the average search cost for the overlay is given by the product of the average cost per hop and the average number of hops per search, which in this case is in the full overlay of N nodes:

$$h \cdot f(N) = \frac{t \cdot f(N)}{k} + \frac{(k-1) \cdot T \cdot f(N)}{k} \tag{3.1}$$

Two-level overlay

In the worst case, we consider a sequential search, where each group is queried until the required data is found. In this case, we need the probability that a query is resolved in the j^{th} group. Since for this analysis any group is equally likely to contain the result [1], this probability is $1/k$. Now, the search cost if the query is resolved in the j^{th} group (l_j) is given by:

$$l_j = j \cdot t \cdot f(n) + (j-1)T$$

The above was obtained from j searches within local groups and the remote jumps between them. Finally, the probability of each cost is used to obtain the average search cost:

$$\left(\frac{1}{k}\right) \cdot \sum_j l_j = \frac{t(k+1) \cdot f(n)}{2} + \frac{T(k-1)}{2} \tag{3.2}$$

If the number of groups k is constant, then the search cost given by (3.1) is dominated by the product of the large remote hop cost T and the average search cost for the total number of nodes N, whereas for (3.2) the term for T is linear and the search cost is dominated by the search cost within groups given by $t \cdot f(n)$.

---

[1] This is a conservative assumption, since in practice nodes will be grouped so that the local group will be most likely to contain the result of a given query.

3.1.3 Application Context

An organization such as a large corporation is a good example of a context for which our two-level overlay is especially well-suited. In such a context, a relatively large number of networked computation elements (nodes) can be interconnected as a structured overlay for the exchange and self-management of data and services. Also, a relatively small and stable number of organizational or geographical divisions exist, such as different departments, buildings, or campuses. Within these divisions, node interactions are frequent, but less frequent interactions between divisions are also necessary. The size of the organization and the relative independence of different divisions make centralized control or knowledge of the behavior of nodes across divisions unsuitable; however, it is reasonable to assume that a common structure, such as the same overlay topology and node ID space is maintained across divisions (however, this assumption can be relaxed, see Section 6.2). Our approach allows for the independent evolution and local management of the overlays (which constitute node groups) in each division, while exploiting the common structure between them to optimize remote interactions.

3.2 Design Principles

The analysis in Section 3.1.2 does not consider how groups are interconnected, but this will significantly affect the functionality and performance of a particular design. The following sections describe our overlay design and analyze its particular characteristics and mechanisms, starting with the mechanism of virtual joins, by which lower level overlays are interconnected and which is the key concept of our design. Some descriptions will be done in terms of the Chord overlay. However, any structured overlay topology can in principle be used for this construction.

3.2.1 Virtual joins

The idea of a virtual join is simple. When a node joins a self-organizing structured overlay, it receives information that allows it to find its place in the given overlay structure. Particularly, it obtains a set of neighbor addresses, which become the links of the joining node within the overlay. In a “real” join operation, the contact information of the joining node would also be disseminated among other overlay nodes and included in their neighbor sets. In a virtual join, however, a node obtains its set of neighbors in the remote overlay, but other nodes’ neighbor lists are not updated. Therefore, the virtually joined node knows its place in the overlay structure, but does not modify this structure upon joining.

Virtual joins can be used to link separately constructed and maintained overlays to form a two-level structure. Each overlay is a local group whose structure and behavior does not differ from a regular flat overlay with a given topology. However, nodes can perform virtual joins in remote groups so that, if need be, they can relay messages to those groups by sending these messages to their remote neighbors. The advantages of these group-level interconnections using virtual joins are the following:

- Because any node can perform a virtual join in any group, inter-group links need not go through designated group heads, thus avoiding potential bottlenecks and single points of failure.
- Messages are relayed according to a common overlay topology, so that the properties of the routing and search mechanisms designed for the original overlay topologies can still be exploited by inter-group communication. This means that routing between groups will be expected to be better (less costly) in the average case than if messages were sent to arbitrary nodes in the remote groups.
- Groups’ local overlay structures are not affected by virtually joining nodes, so that any number of nodes may perform virtual joins in a particular group.

Of course, there are limitations to the virtual join mechanism. While any number of nodes may virtually join a particular group, a particular node is limited in the number of groups that it can join. This is because it must keep references to all its neighbors in all of these remote groups. Since for the context we consider (Section 3.1.3), the number of nodes is expected to be much larger than the number of groups, this is not a significant problem, and nodes can join every remote group to create a fully connected high-level overlay.

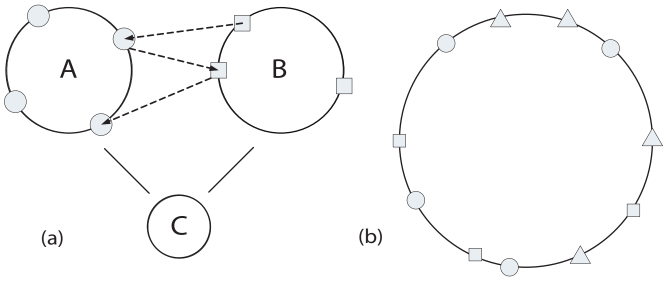

Figure 3.1: Two-level overlay organization for Chord rings. a) Shows the fully connected high-level network of three groups A, B, C, and the individual lower-level groups A and B, as well as some of the successor relations between nodes in these groups that realize the high-level link. b) Shows an integrated view of the network, where for each different shape there is a directly connected group.

Virtual joins in Chord

In Chord's ring-based topology, a node joins by contacting any other node already in the overlay, from which it obtains references to its successor and predecessor in the ring, as well as a number of extra references (chords or fingers) to allow for faster traversal of the ring. References are kept in a data structure called a finger table. Successor and predecessor references are enough for successful routing in Chord. In our two-level overlay design for Chord, local groups are organized into independent Chord rings. For each group, nodes' finger tables contain only local nodes. Using a single identifier space for all nodes (that is, all nodes obtain their identifier in the same range), any node in one group can perform a virtual join in other groups, querying the remote group for its successor in that group (in practice, we use the predecessor, for reasons explained in Section 3.3; for now, assume the successor is used for explanation purposes). Figure 3.1 shows the configuration of a three group overlay and the intra and inter-group successor-predecessor links.

When a new node joins the system, it will normally obtain its identifier and join its local group (assigning nodes to groups is a system specific issue, and must be defined beforehand, either by a system administrator or by some automated mechanism). It will then perform virtual joins with this same identifier in each of the other groups that make up the system. As in normal Chord joins, a virtual join only requires knowledge of one existing node in each ring. Each node will then store its own finger table as well as its successors in each of the remote groups it has virtually joined. Since we are working under the assumption that the number of groups is small compared to the number of nodes within each group, and is relatively stable or static, the additional space needed for the external references (one per remote group) will be smaller than the local finger table.

3.2.2 Inter-group Routing and Search

Given the dynamic nature of existing overlay routing protocols, we assume that, given a message addressed to an overlay identifier i, the node responsible for this identifier within a particular group g ($s_g(i)$) will always exist and can be reached by the routing mechanism. In order for search to make sense, however, the overlay must support a higher layer abstraction that will evaluate i, or the message associated with i, and decide whether or not this message can be handled at $s_g(i)$. For example, if the overlay supports a DHT, as is commonly the case, i will be the hash value of a key. Node $s_g(i)$ will be able to resolve a query for a particular key if an object or objects with that particular key are stored at the node.

The basic idea behind search in the two-level overlay is the following. Normally, routing is done within local groups only, that is, the routing protocol will find $s_l(i)$, the local node that is responsible for i. Only if $s_l(i)$ is unable to handle the message delivered to it will a remote group be queried, via the remote neighbor(s) of $s_l(i)$. This justifies the intuition of the earlier claim that search between groups will be expected to be less costly in the average case using the virtual join links as compared with arbitrary links: local routing uses the overlay structure, so that the destination node is the closest node in the local group to the message identifier i. Thus, its neighbors are likely to be $s_r(i)$, the node directly responsible for i in the remote group, since they are expected to be close to i in the structure.

The two-level overlay network provides a simple abstraction to the layers above to enable routing and search, through the operation **resolve**(*identifier*, *group*). Given a data identifier i, this operation returns $s_{group}(i)$. If the group parameter is omitted, the lookup will take place in the local group. Two wildcard values are accepted for

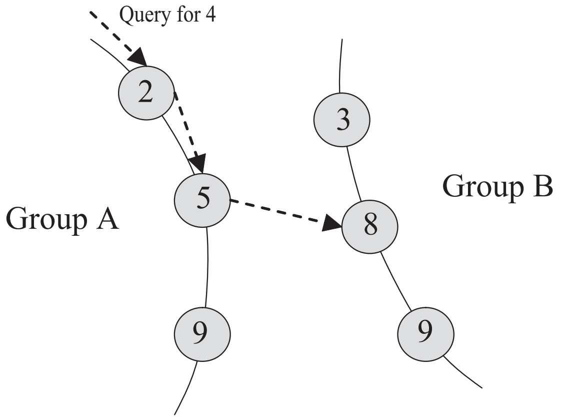

Figure 3.2: Routing example in two-level Chord overlay

this parameter for inter-group routing: ANY, which is used to route to a node on any group, starting with the local group, and ALL, which is used to route to nodes on all groups. Both values first initiate routing normally to $s_l(i)$. For ANY, a message is propagated further only if it cannot be handled on that node. For ALL, the query is sent in parallel to every other group. Note that in both cases, because the access point to remote groups is a neighbor of $s_l(i)$, routing in remote groups is expected to be resolved in better than average number of hops, as explained above.

Figure 3.2 shows an example of routing between two Chord rings. The query for id 4 is routed to $s_A(4) = 5$. If the corresponding query cannot be handled there, it is routed to $s_B(5) = 8$, which in this case is also $s_B(4)$, and is thus responsible for it in this group.

3.2.3 Overlay Maintenance

Most existing structured overlays are self-maintaining, in the sense that they are designed to autonomously deal with node departures and failures, in addition to node joins. In Chord, failures are recognized by periodically probing the entries of the finger table to find if the referenced nodes are still up, or if others have recently joined to take their place. Chord has self-repair mechanisms that allow it to fix broken finger table entries by queries to successive neighbors.

The state of remote links obtained by virtual joins in the two-level overlay can also become outdated and require repair. However, because of the high cost that we assume for inter-group communication, and because of the redundancy in inter-group links, our design does not prescribe periodically checking remote neighbors. Instead, the state of a particular link will only be checked when an actual message needs to be sent to a remote group via that link. There are two cases in which a node's remote links become outdated:

- If the remote neighbor of a node has left the overlay or failed, the node will no longer have a link to the remote group. The node can then simply use the remote neighbor of one of its local neighbors (say, its predecessor) to relay the message. Notice that in this case, the remote node may not be as close to the destination as expected for normal inter-group routing, but this only affects the performance and not the result of the search. The node can then perform a new virtual join to update its remote link.
- If new remote nodes have joined so that the link stored at some node no longer corresponds to its actual remote neighbor, effective communication may still take place, though performance will be affected. When a message is sent to a remote neighbor, that neighbor can respond with its local neighbors. That way, the sender may check if any new neighbors exist which can replace its existing entries. In Chord, for example, each time a message is sent to a remote successor, it can reply with its local predecessor, which may be a better remote successor for the sender. This is the same mechanism used in Chord's stabilization protocol, except that it is not meant to be executed periodically as in Chord.

3.3 Evaluation

We conducted experiments aimed at verifying the intuition that our two-level overlay with virtual joins has a better than average inter-group search overhead, measured in number of hops, than a two-level overlay with group heads. The experiments were run on 64 nodes, which were divided randomly into two chord rings. The probability for each node to be in a particular group was chosen from among the values of 0.5, 0.7,

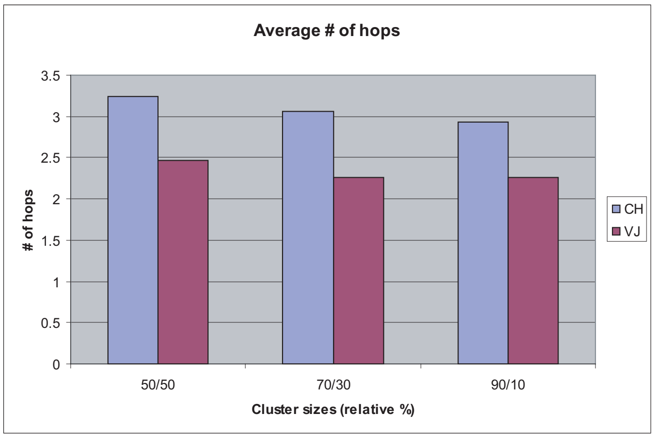

Figure 3.3: Average number of hops in remote group searches for group-head (CH) and virtual join (VJ) connections

and 0.9. The reason for this was to examine the effect of the relative size of the groups on the results.

Each node stored a unique value, and each of these values was the subject of an inter-group query. The number of hops of each query in the destination group (where the two structures differ) was measured. Figure 3.3 summarizes the result for the average number of hops on all queries for both structures, with the different relative group sizes. Notice that the number of hops is small in both cases, since search in Chord is $O(\log n)$, and precisely because of this, the difference is significant: a difference of 1 in the number of search hops means an order of magnitude difference in the number of nodes for which this search efficiency is achieved.

Figure 3.4 more directly confirms the intuition that inter-group searches will be started close to their required destination when using virtual joins. The lines plot the percentage of queries that were resolved in up to one hop (which means the first remote node queried either resolved the query directly or was the predecessor of the resolving node). For groups of roughly equal size, this is equivalent to about half of the queries. The percentage drops as the relative difference between the group sizes increases because, when searching from the smaller group, the number of entry points to the larger group decreases. Figure 3.5 clearly shows how this metric improves for searches from large to small groups, rather than the other way around. For the structure

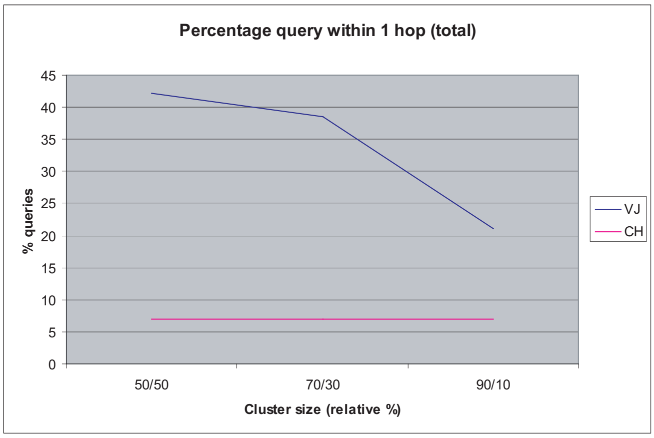

Figure 3.4: Percentage of total queries within 1 hop in remote group for group-head (CH) and virtual join (VJ) connections.

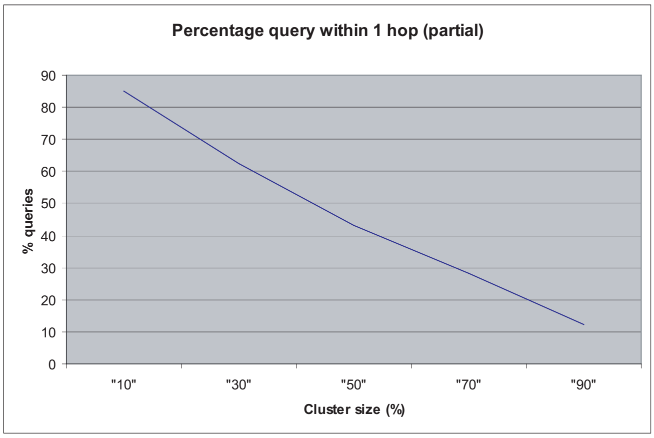

Figure 3.5: Percentage of queries in each group within 1 hop, according to the relative size of the group.

that uses group heads, this percentage is constant, because the only queries that can be resolved in up to one hop are those that can be resolved by the group heads themselves or by their immediate successors.

One important observation gathered from the tests is that using a nodes' remote predecessors as inter-group links instead of their successors produced better results. This is due to the unidirectional routing used by Chord and the fact that it is possible for there to be remote nodes between the key value i being sought and $s_r(s_l(i))$. In this case, the remote neighbor will not be responsible for the key and must route the query practically all the way around the ring to come back to the required node. By using the predecessor as the remote neighbor, the likelihood of the above to happen is reduced because either the remote node precedes key i and routing can be concluded in a few hops, or the node is a closer successor to i than is $s_r(s_l(i))$.

Chapter 4

Content-based Notification Service

This chapter describes the design and implementation of the notification service for subscription management and notification dissemination targetting highly dynamic Grid environments. The design is centered around the Web Services Notification standards, which provide a common platform independent interface for communication, and is based on the Meteor distributed and decentralized architecture that supports content-based, loosely coupled communication. The self-managing two-level Chord overlay described in Chapter 3 supports the whole system infrastructure, providing separate levels of coupling within and between separately communicating node groups for enacting different kinds of tasks or workflows. To further support the efficiency and scalability of our approach, we design self-optimization mechanisms for reducing the number of messages transmitted by the system. These optimizations are meant to alleviate the overhead of notification flows.

4.1 Service Design

The notification service is designed as a distributed and descentralized notification broker. Each of the nodes within the system is a peer that implements the NotificationBroker (NB) interface (see Section 2.2), so that an external client can interact with any of them. Thus, the whole system acts as a single NB, as illustrated in Figure 4.1. This is important because the interface is not a bottleneck, and the system has no single point of failure. Service providers participate by making nodes available to the notification system, and can in turn make use of the system through these nodes. A peer-to-peer design avoids the need for centralized control and gives the service providers the flexibility to join or leave the system at will.

Through this interface, clients and brokers realize the message exchanges defined

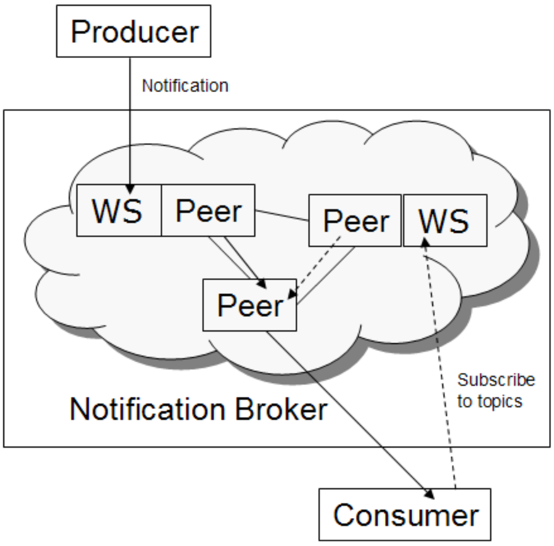

Figure 4.1: Layout of the broker network. Matching subscriptions and notifications will be routed to the same rendezvous node, which will perform the matching and relay the notification.

in the WSN specifications, using XML messages that represent subscriptions, notifications, etc. Topic expressions are used by the notification service as identifiers for these messages, and provide the means for matching between them. The following chapter further explains the use of topics in our system.

4.1.1 Content-based Topic Expressions

Unlike a purely topic-based system, such as WSN with topic filtering, topics in the notification service are meant to be content-based. The notification service uses topic expressions that extend the concrete dialect of WST (see Section 2.2) to encode content-based descriptors that correspond to attribute/value pairs. This means that a topic expression is no longer an atomic unit that corresponds to a path in a hierarchical topic space definition, but a sequence of identifiers, each of which is taken as an attribute value from a multidimensional information space.

To observe the difference, consider a weather monitoring application that subscribes to sensor data. In this example, the application may define the information space with three dimensions: state, city, and temperature. A topic for a notification in this system might then be `weatherService:NJ/Piscataway/80`, while a subscription could

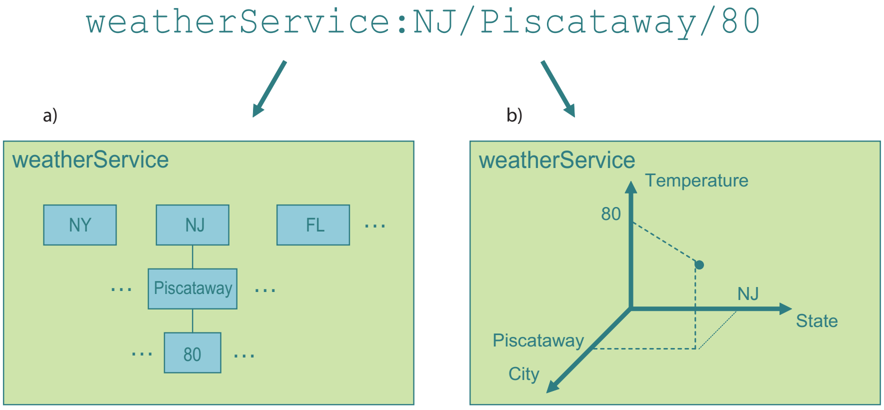

Figure 4.2: Construction of a topic expression from a) a hierarchical topic space; and b) a multidimensional information space

be `weatherService:NJ/*/>75`. As Figure 4.2 shows, defining a hierarchical topic space for this type of topic expression would not be practical, since individual topic identifiers would be needed for each geographic location, and, worse still, for each possible numeric value. By encoding content-based identifiers in topic expressions, our approach seeks to remain as close as possible to the notation defined in WS-Topics for topic expressions, both to support applications that implement this standard and to simplify content-based indexing while still taking advantage of the latter's expressive power. We avoid the use of content filtering as defined in WSN (see Section 2.2), which is more powerful and flexible, but also more costly to implement, as it requires parsing of the payload content.

4.1.2 Messaging Model

The system uses Meteor's rendezvous-based messaging model in which matching messages "meet" at some node within the network, referred to as a rendezvous node. The matching and routing of messages to service nodes is done by parsing topic expressions and using Squid to map their constituent values to the node identifier space, as described in Section 2.3.1. Squid ensures that matching topics will be routed to at least one common rendezvous node. We use Meteor's reactive behaviors embedded in message requests to encode the operations defined by the notification interfaces (subscribe, notify, etc.). The application of this model to these operations is described below:

Subscribe and Notify

As a notification producer (NP), the distributed broker accepts subscriptions from clients. Subscriptions handled by the notification service must contain a topic expression within the FilterType element, as explained in Section 2.2. When a subscription message is received by any one of the broker peers, its topic expression is decomposed into its constituent values and mapped to the node identifier space. Since a subscription topic can contain wildcards or ranges, the subscription may span multiple topics, which may correspond to one or several nodes. Each one of these nodes stores the subscription, keeping it until a termination time that may be included in the subscription message is reached, or until the subscription is cancelled by the client.

To produce a unique identifier for the subscription, the entire topic is hashed, along with the consumer endpoint reference. This ensures the differentiation of subscriptions for the same topic or topics from different consumers. The unique identifier is appended to the topic expression, which is returned to the client as the subscription reference to be used for subscription management.

Notifications are handled by brokers, acting as notification consumers (NC's), upon invocation of the Notify method. The procedure is similar to that of a subscription. If the notification's topic expression is singular, in the sense that it contains neither wildcards nor ranges that span multiple topics, then the notification maps to a single rendezvous node within the network. If a subscription for that topic exists at the rendezvous node, then the consumer reference is extracted from the subscription record stored at the node and used to connect to the client and relay the message. Figure 4.1 also illustrates this rendezvous process. If a notification is identified by a topic expression that spans mutliple topics, then the notification isn't routed to a rendezvous node as above. The reason for this is that there might exist multiple rendezvous nodes for it, a number of which may store the same matching subscription, resulting in the same notification being relayed to a consumer multiple times. Instead, the interface node that received the notification queries the network for subscriptions, and then directly relays the notification only to each different consumer reference from the subscriptions it receives. A similar procedure is also used for demand-based publishing.

Subscription Management

As was mentioned above, a subscription reference in the notification service consists of a topic expression and a unique identifier. The functions defined by WSN for the `SubscriptionManager` and `PausableSubscriptionManager` interfaces depend on this subscription reference because, given that nodes can enter and leave the system at any time, subscriptions should not be tied to a particular rendezvous node. Recall from Section 2.2 that in WSRF an endpoint reference can be used to interact with a subscription resource directly. However, because of the above and the fact that we do not assume that subscriptions are WSRF resources (as this is optional in the standard), such endpoint references are not used. Thus, the topic expression is always used to route the requests to the node(s) at which the corresponding subscription is currently stored. Once at these nodes, the subscription's unique identifier is used to quickly obtain the particular subscription and execute the appropriate action.

Pull-Style Notification

Pull-style notification in the notification service is done in a very similar way to the way regular subscriptions are handled. The only difference is that, when a pull-point creation request is made to a broker, a message repository is also created at each rendezvous node where the subscription is stored. Notifications are stored in these repositories rather that being relayed directly to the consumers. Finally, when a consumer invokes the `GetMessages` command on a broker, it queries the network with the subscription reference to obtain the notifications stored at the repositories, constructs a single response with all of these notifications, and sends them back to the client.

Demand-based Publishing

Publisher registration occurs in the notification service in exactly the same way as a subscription. The registration topic is used to route a registration message to a node or nodes in the network. In order to accommodate demand-based publishing, however,

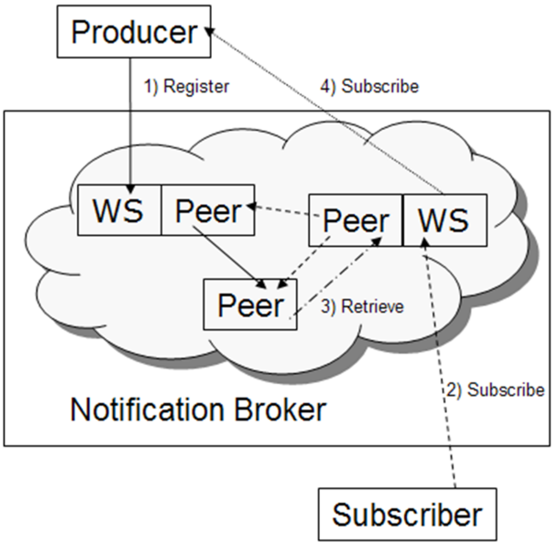

Figure 4.3: Publisher registration and subsequent subscription. Notice that a subscription and registration do not necessarily span the same nodes, but as long as they overlap at some node, the registration will be retrieved.

the procedure for a subscription detailed above must now include a query for publisher registrations that match (or, rather, overlap with) the subscription's topic expression. If such registrations exist, then the NB that received the original subscription subscribes in turn to the producer(s) for the topic(s) given in the registration(s). Figure 4.3 illustrates this mechanism.

4.1.3 Messaging Optimization Mechanisms

The number of messages sent within the system can be reduced at the notifications level, which is important because any reduction in the number of messages leads to a reduction in the overhead involved in packaging and delivering each individual message, and to an improvement in scalability. In the case of Web Services, this overhead is incurred mainly by XML and SOAP headers. In addition, the messaging within the JXTA peer-to-peer framework [30] that supports our current overlay implementation also adds considerable overhead. To see how much bandwidth is actually consumed by overhead in one implementation, a sniffer program was used to capture the packet flows between the nodes in the network for notifications. For messages between network nodes, the combined overhead of XML and JXTA for each message is just over 3.5 KB, which amounts to about 28 Kbps in a message flow of one message per second. The following mechanisms are used by the system to reduce the number of individual messages in the network.

Grouping of Notifications by Buffering

This optimization is meant to reduce the flows of small and frequent notifications. A simple way to deal with these notification flows is to buffer and group several notifications within a single notification message, a mechanism provided by the WSN XML schema. This way, the headers that would have been transmitted with every individual message are reduced to a single header on a grouped message. The optimum time and degree to apply buffering, however, will be different for different message flows, and thus it is worthwhile to equip the system with logic that allows it to autonomously determine the most appropriate level of message aggregation based on high-level constraints.

Without application-specific considerations, messages can be grouped based on two criteria. The first is on messages that correspond to the same topic, and the second is on messages that match the same subscription. These criteria are not necessarily the same, since, depending on how broad a subscription is made (with wildcards or ranges), several different topics may match a single subscription. The system can benefit from applying both criteria, since grouping based on topic equality can be done when messages enter the system at an interface node, which doesn't necessarily know about subscriptions for that topic, and then subscription-based grouping can be determined at the rendezvous nodes. The mechanism, however, is the same in both cases, so we will describe grouping based on topic equality.

The mechanism for grouping and packaging of notifications is as follows. Each interface node keeps a separate buffer of messages for every topic it receives (garbage collection can be employed to eliminate buffers for which no messages arrive for a period of time). Each buffer is configurable by setting the length of the period during which messages are accumulated. This buffering level is determined by managers associated with each buffer, the design of which is described below.

If the buffering period is determined only with respect to bandwidth utilization (the number of messages), then the solution is trivial because a higher buffering level (more messages grouped together) always increases the saving achieved. If a limit is set on the buffering period, according to the maximum latency allowed for each individual message, then the solution would always be set to this limit. However, a more balanced solution should consider the tradeoff between bandwidth utilization and message latency. An optimal point can be found between a buffering period of zero (minimum latency, maximal bandwidth consumption) and one equal to the maximum allowed latency (highest buffering level, minimal bandwidth consumption). This is the range used in Equation 4.1 below, although the reciprocal of the incoming rate is used as a lower bound instead of zero (any period set smaller than the incoming message period would result in no buffering). Instead of manually assigning a weight to each extreme, a dynamic solution is determined based on the relative size of the payload with respect to the total message size (Equation 4.2 below). The rationale behind this is that the relative saving in bandwidth is greater for small messages because the overhead constitutes a larger fraction of the total data sent, whereas for large messages the overhead becomes relatively insignificant. In the former case, there is greater payoff for sacrificing latency, and thus buffering should have a larger weight. For the latter case, the reverse is true. Finally, the buffering period is calculated by obtaining a value within the range determined by the weight, using Equation 4.3. If the incoming rate is very low, with a period higher than the maximum latency, then Equation 4.3 is not used, and rather the period is set directly to zero.

$$
\begin{aligned}
range &= maxLatency - avgIncomingRate^{-1} && (4.1)\\
weight &= \frac{avgPayload}{overhead + avgPayload} && (4.2)\\
period &= maxLatency - weight \times range && (4.3)
\end{aligned}
$$

Demand-based Notification Relay

Ideally, notifications should not be relayed by the notification service if no subscribers exist for them. Demand-based publishing, explained in Section 4.1.2, is WSN's provision for dealing with this issue. However, demand-based publishing depends on producers registering their topics with the notification broker, which particular publishers may not choose to do or may not be able to do if they do not implement the NP interface. To further optimize messaging, the system implements a mechanism which is similar to that of demand-based publishing but that is based on the topics of individual notifications. The idea is that interface nodes should determine when not to relay notifications to rendezvous nodes based on the existing subscriptions.

Unlike publisher registrations that define the topics that will be produced beforehand, an interface node has no way of knowing which topics it will have to handle. Registering for every topic received would also be inefficient. Thus, the mechanism devised is implemented as follows. Each interface node keeps subscription caches associated with particular topics. If there is no cache associated to a particular topic when a notification for it is received, the interface node queries the network for subscriptions for that topic. If any are found, they are placed in the subscription cache, which is marked as empty otherwise. Subsequent notifications with the same topic will only be relayed if the corresponding subscription cache is not empty. To avoid making a query for every topic received, locality is exploited by checking a topic against all cached subscriptions. New queries are only made if no subscriptions exist in these caches (note that for these topics, the notification is relayed in any case).

Meanwhile, at the rendezvous nodes that responded to the query, a temporary registration is kept of the interface nodes and their corresponding queries. This ensures that if a subscription did not exist at the time of the query, a matching subscription made thereafter can be made known to the interested nodes, so that a notification for which a subscription exists is not dropped. The same happens for the cancellation of subscriptions. Because they are potentially more numerous than publisher registrations, rather than keeping these registrations indefinitely, they are deleted once they are used. Thus, interface nodes must requery the network once a cache for a particular topic becomes empty.

The messaging overhead of this mechanism for each topic are the query and its corresponding response (2 messages), as well as one message per update of a subscription

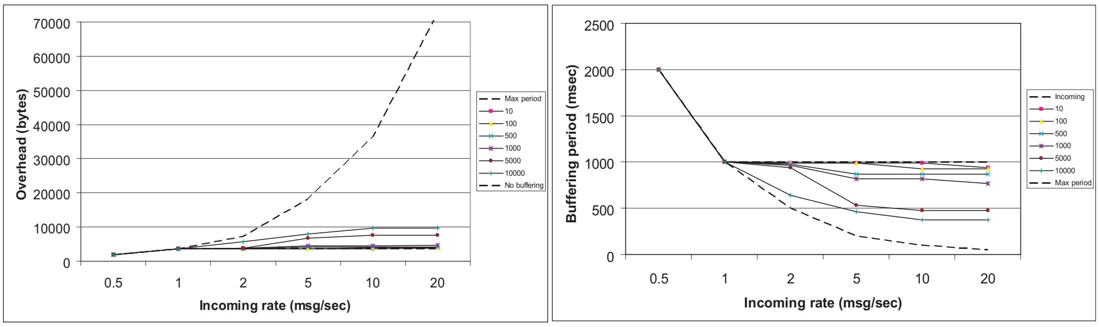

Figure 4.4: Results for buffering with different incoming rates and payload sizes. Lines correspond to the different payload sizes, bracketed by the minimum and maximum values for each measure. Left: Overhead bandwidth, grows with payload size. Right: Buffering period set, decreases with payload size.

or its cancellation. New queries are only triggered after cancellations. Thus, the overhead is small and can easily be made up when large notification flows are not relayed while no subscriptions exist, unless rates of subscriptions and cancellations are in the same order as the rate of notifications.

4.2 Evaluation

As a proof-of-concept of the optimization mechanisms, experiments were conducted to observe the buffering behavior and resulting overhead for message flows of different incoming rates and payload sizes. The experiments were set up with a maximum latency allowed for messages at each node of 1 second. Figure 4.4 plots the results. Notice that savings in bandwidth utilization are substantial, even though buffering periods are distributed within the range of allowable latencies. The lowest buffering period set in this case is 373 seconds for the message rate of 20 messages per second and 10000 bytes per message.

For irregular notification flows, possibly originating from several producers publishing notifications on the same topic at different time intervals, several complications are possible, such as short bursts of notifications at high rates, high variability in the incoming rate, and concurrency. A number of mechanisms were used to reduce the sensitivity of the system to these conditions. To emphasize the self-managing aspect of the system, the use of fixed low level parameters was avoided. For example, instead of

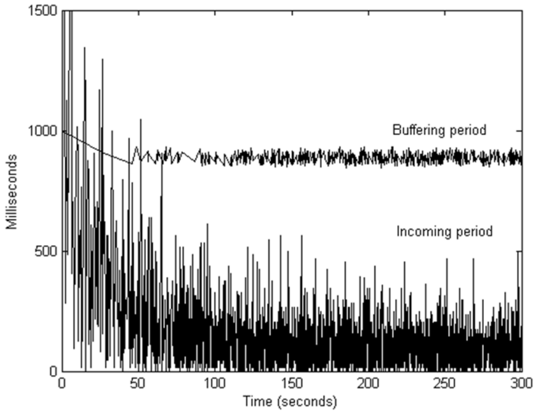

Figure 4.5: Change in the buffering period with highly variable incoming periods

using a fixed threshold for the minimum change allowed for the buffering period, the threshold is calculated dynamically based on whether or not the change in buffering would cause at least one message more or less to be buffered at the current estimated incoming message rate. This new parameter (the change in the number of messages for which a change in period is allowed) is at a higher level and is more meaningful than the period alone.

To test the behavior of the system under these conditions, an interface node was set to receive messages with the same topic from 32 different producers, each one of which sent messages of random payload size between 10 and 500 bytes at random intervals of up to 5 seconds. The combined effect of these notifications produces a high message rate, with high variability. Figure 4.5 shows the changes in the buffering period during the time of the test.

Chapter 5

Decentralized Clustering Analysis for Self-Monitoring

5.1 Motivation

In Chapter 1.2, we introduced the problem of self-monitoring of distributed systems as an immediate application of the content-based notification infrastructure, for several reasons. First, the content-based publish/subscribe communication pattern allows for selective subscriptions to situations of interest in a monitored system, given a meaningful representation of monitored attributes. Second, the content-based routing mechanism itself (described in Section 2.3), used to deliver messages based on attribute profiles, exhibits clustering properties that can support in-network, decentralized data analysis for proactive monitoring (the detection of indicators or patterns that precede and point to a failure or other event of interest). Additionally, the application of monitoring can be conceived as a generic service that runs on top of the notification service.

Proactive monitoring involves in some way online data collection from the target system and an online analysis of this data, which is typically centralized and offline. However, decentralized, in-network analysis is desirable for several reasons. First, online analysis can produce results in a timely manner, so that corrective actions can be more responsive to the events in the network. Also, offline analysis requires storage, communication, and computation resources that are external to the peer network and that can be costly to operate and maintain. Data loss during transmission to the central analysis engine can be another concern. It is more difficult to recover from the loss of raw data after it has been transmitted to the central repository than it is for data that has been processed in the network, since processed data is more readily identifiable and can be saved for a longer time. Finally, data privacy is an important issue, especially when raw data may convey information that individual stakeholders in the system do not want to share outside their local scope. Processed data can hide this information and only convey information that is relevant to the monitoring task.

This chapter describes a decentralized and online mechanism for the proactive self-monitoring of large peer-to-peer systems such as device networks (e.g. networked appliances, document processing devices, network routers, wireless devices), server farms, and compute clusters, where the peers have similar components and operational behaviors. It is assumed that peers in the system represent their behavior and operational status using commonly known attributes and periodically report this information as semantic events over the content-based messaging and notification substrate. External entities (management systems or meta-workflows that depend on monitoring information) can subscribe to these low-level events or to high-level events that notify of anomalies or system trends obtained from our decentralized in-network analysis engine.

5.2 Other Self-Monitoring Systems

In general, methods for self-monitoring are based on characterizing normal system behavior and then detecting deviations from this normal behavior. A number of different approaches have been proposed and developed, most of them specialized to different application contexts. These include system-specific approaches that exploit the particular characteristics of the data acquired during monitoring.

For example, in [31], a method for analyzing the correctness of the Windows registry based on extrapolating from samples of known bad registry entries is described and used for detecting real errors in existing registries. Other more general approaches include learning decision trees from system logs that are then used for diagnosing system failures, i.e. finding the causes of failures, as described in [12]. Automata constructed from normal user request traces in web applications are used in [26] to recognize abnormal interactions of application components that are the result of or lead to faults. The work described in [27] also models normal application component interactions based on user requests, but does so indirectly by measuring the intensities of data flows between components and modeling invariant relationships between input and output flows at each component. Note that all of these methods depend on offline training to develop the respective models used for runtime monitoring.

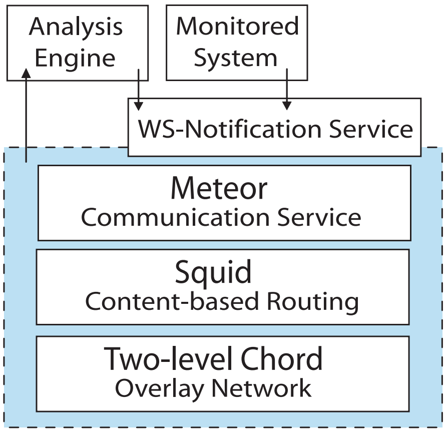

Figure 5.1: Data flow for decentralized monitoring and analysis

The approach in [8] gathers data from HTTP access logs of web applications online and keeps track of page access frequencies to identify and report anomalies to administrators using special visualization tools. Administrators can then interpret and deal with these anomalies using application-specific knowledge. Our monitoring mechanism also uses data collected online to recognize and report anomalies. However, unlike the approaches described above, data analysis is carried out in-network by peer nodes in a decentralized manner.

5.3 System Description

Figure 5.1 shows the elements of the monitoring service within the workflow management framework. Peers publish semantic status events through the notification service, and these events are received by the analysis engine, along with any external subscribers that are interested in specific events or event types. The analysis engine makes use of the status events to detect anomalies in the behavior/state of peers by finding the clustering of these events, in space and time, based on the values of their attributes.

Clustering analysis is a well-known data mining technique. In this type of analysis, the similarity of data points in a coordinate space is established by grouping them into clusters, where points in each cluster are relatively close together, according to some notion of distance. Anomalies are detected as singular data points or isolated clusters. For example, if a measure of peer load is encoded into the events reported by peers, these events should exhibit high clustering if the network is load-balanced. Underloaded or overloaded nodes will show up as anomalies in this analysis. Multiple dimensions can be analyzed simultaneously, so that, for example, unusual resource consumption under similar workloads can be detected as an anomaly.

There are two types of events that result from the clustering analysis: status updates that are recognized as anomalous (points that don't belong to detected clusters) and cluster centroids (points with minimum average distance to the points in their cluster). These events are also published using the notification service to be received by external subscribers. These subscribers can respond to anomalies as well as further analyze centroid data. The idea with the latter is that the cluster centroids represent a significant reduction of status data with respect to the raw data and can be more easily analyzed for trends and correlations between the different monitored attributes, without significantly reducing the accuracy of this analysis.

5.3.1 Monitoring Information Spaces

The common attribute sets that respresent peers' monitored operational status can be viewed as one or more multidimensional information spaces, as illustrated in Figure 5.2. Each dimension in such an information space corresponds to an attribute that can be monitored and reported by a peer node. Consequently, an event that represents the status of a peer at some point in time corresponds to a point in the information space. Further, it is assumed that the value ranges of the attributes are defined so that values that are close together along that dimension of the space represent states that are similar in terms of the corresponding attribute. This is normally the case with quantitative attributes, and can be made so with an appropriate encoding for non-quantitative attributes as well (in the case of completely unrelated attribute values, two values are only considered close if they are equal). Given this assumption, points that are close in the multidimensional space will also correspond to similar status descriptions.

Using the representation described above, if the status events of all the peers are plotted as data points in their corresponding information space, clusters of data points will indicate peers with similar states, while outliers will indicate possible anomalies, and thus, possible precursors of violations or other conditions (failures) that may need attention. A similar analysis can also be made across data points generated at different times, to analyze similarities, differences, or trends in the behavior/status of peers over time.

5.3.2 Decentralized Clustering Analysis

Cluster recognition and analysis of the type described above is a well-known problem in the field of data mining, and k-means clustering is a widely-used procedure for which distributed algorithms have been devised [7, 14]. Traditional k-means is an iterative process that calculates the coordinates of k centroids (points with minimum average distance to the points in their cluster) for a number of data points. Both k and an initial estimate for the centroids must be given as input, and the algorithm successively associates data points to their nearest centroid and recalculates centroids until they do not change significantly from one iteration to the next.

The distributed procedures cited above partition data points uniformly among processing nodes, each of which runs the k-means procedure on their subset of points. Nodes then repeatedly exchange centroids with neighbors and rerun the algorithm until a consensus is reached. Note that these approaches do not consider locality while distributing data points to nodes, and as a result, convergence can be slow and cause the algorithms to be unsuitable for online clustering.

The decentralized clustering algorithm used here is similar to the distributed algorithms described above in that it also partitions data points across the processing nodes. However, this formulation exploits spatial locality in the information during distribution to facilitate the clustering analysis. The key aspect of our decentralized clustering formulation, as illustrated in Figure 5.3, is a dynamic one-to-one mapping of distinct continuous regions of the multidimensional information space to processing nodes. This mapping is achieved using the SFC mapping described in Section 2.3.1. The correspondence of a region to a node implies that any point within the region must also be mapped to that node. The continuity of the regions ensures that each node will handle similar data points, as explained above. Finally, the mapping is dynamic

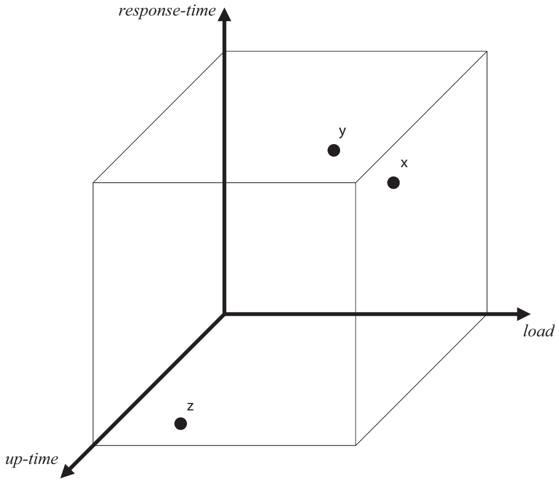

Figure 5.2: Example of an information space with three attributes. Points x and y in the space are close and thus considered similar, while point z is considered different.

because the peer network intended to host the processing nodes is itself dynamic. As a result, regions assigned to nodes may be split or merged over the course of time, and not all regions will be of equal size.

Cluster recognition and anomaly detection is based on the principle illustrated in Figure 5.4. The key idea is that, given a fixed number of data points with a perfectly uniform (or random) distribution, the point density (the total number of points divided by the volume of the information space) would be constant throughout the space (see Figure 5.4a). Thus, there is an expected number of points per region of the information space that can be calculated based on the size of the region and this constant point density. Clustering is recognized when a region has a relatively larger point count than what is expected for that region (see Figure 5.4b). Conversely, if the point count is smaller than expected, the points in the region are not part of a cluster (which must exist elsewhere) and can thus be treated as anomalies. Appropriate threshold values can be set to fine tune the sensitivity of cluster and anomaly recognition, depending on the degree of clustering expected in the status updates of a particular system.

Region boundaries can affect analysis results. For example, a particular region may contain a few points from a cluster that lies in a contiguous region, so that they are falsely labeled as anomalies. This can be seen in the lower left quadrant of Figure 5.4b. Because of these boundary conditions, nodes need to be aware of their neighbors in the

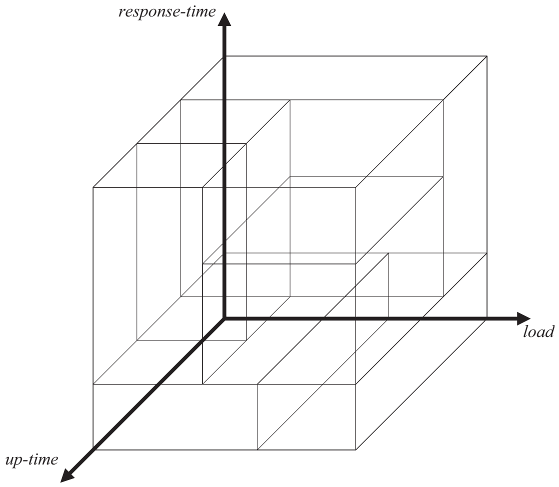

Figure 5.3: A division of the information space, where each region corresponds to a different node.

information space (nodes responsible for neighboring regions). We approach this in the following way. When a node recognizes that its region contains a cluster, it calculates a centroid for that cluster. We assume a minimum distance d exists such that two points are considered similar if their distance from the centroid is less than or equal to d. If a region boundary is found within this distance d of a centroid, then the node responsible for the neighboring region will be notified of the existence of the cluster and sent the location of the centroid. Using this information, nodes can associate their points with nearby centroids in remote regions, if possible.

Because the size of the region of a particular node is dynamic and depends on the total number of nodes across which the information space is divided, there may be more than one cluster in a given region. The distance d is used to estimate the number of individual clusters (k) that could be recognized in a region of a particular size. If $k > 1$ for a particular cluster, then the k-means procedure is applied locally to find the relevant centroids.

5.4 System Operation

Self-monitoring peers within the framework go through three phases: setup phase, data generation/collection phase, and analysis phase. Each peer runs the setup phase first when it joins the system, and then every time that the joining or leaving of a neighboring

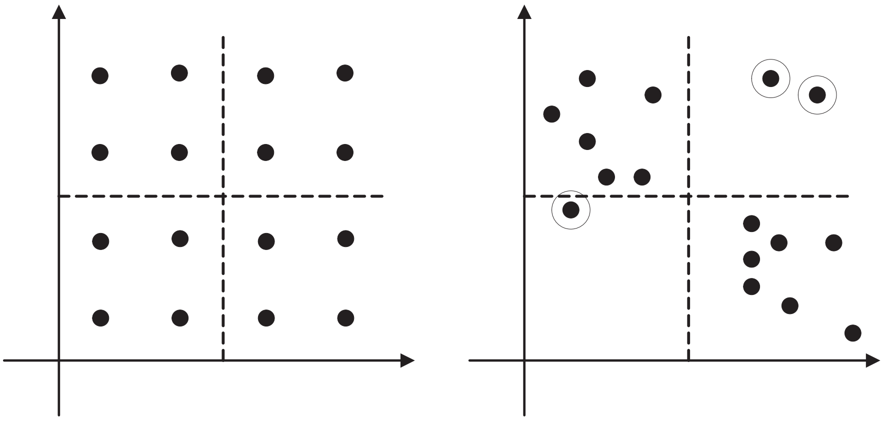

Figure 5.4: (a) Uniform distribution of data points in the information space. (b) Point clustering among regions; the circled points are recognized anomalies.

peer affects the mapping of data points to it. Once the setup phase has been completed, the data generation/collection phase runs continuously, and is only interrupted when the analysis phase is triggered. The specific tasks in each phase are described below.

When a peer joins the system, it becomes a processing node for the clustering algorithm and must thus be assigned the region of the information space for which it will be responsible. This assignment constitutes the setup phase of the monitoring algorithm. The algorithm depends on the specific mechanisms used by the underlying DHT abstraction to map the information space to peer nodes. In Squid, for example, a joining node is assigned a specific ID in the Chord index space using the Chord join protocol. Since this node will be responsible for the span of the index space between its ID and the ID of its predecessor, the corresponding region of the information space can be computed using a reverse SFC mapping. The node will then receive all messages (events, subscriptions, notifications) that correspond to this region of the information space. Note that the successor of the joining node will also have to rerun the setup phase because the region previously assigned to it is now split between itself and its new predecessor.

Each node in the system calculates its expected data point count during the setup phase. In addition to the size of its region, the node must have an estimate of the total number of peers in the system in order to approximate the total number of updates that will be generated by the peers during the data generation/collection phase. In a dynamic peer-to-peer network this is not trivial, but there are existing methods for obtaining peer counts, even in the face of high churn [5]. Futhermore, we assume that churn in the device and server networks that our approach targets is low (a relatively constant or slowly changing total number of peers is maintained), so that the overhead in obtaining and maintaining an estimate of the peer count at each node will be low.

Once a peer has joined the system and is operating normally, it will periodically generate and transmit status updates as semantic events that represent data points in the information space. The content-based messaging substrate will route these events to the peer nodes responsible for the regions of the space that contain these data points. The node receiving an event will store it locally, increment its counter of events received, and generate any relevant notifications if it has any matching subscriptions. The count will be used in the analysis phase for clustering.

Since the clustering analysis uses data point counts, the synchronization of event generation/collection periods becomes an issue. We assume that all peers in the system use the same period length for generating events. Because generating periods will nonetheless be staggered, a receiving node will listen for at least two periods before examining its point count. As a result, a particular node will receive at least one update from peers with status mapping to it before each analysis phase.

The analysis phase is essentially the execution of the decentralized clustering algorithm presented in Section 5.3.2. Once a peer has found a cluster or an anomaly, it uses the notification infrastructure to publish this data, so that interested peers or other entities can use this information to trigger appropriate corrective actions, analyze trends, etc.

5.5 Functionality and Effectiveness

The proactive self-monitoring framework and the decentralized clustering algorithm presented in this paper have been experimentally evaluted using simulations. In the experiments presented in this section, the status updates of a given network of peers were simulated by generating semi-random points in a two-dimensional information space.

This could correspond to, for example, a device network that produces measurements of load (processed documents, transactions, serviced web pages) and resource consumption (toner, CPU cycles, bandwidth) for each peer. Note that the evaluations in this section focuses on the effectiveness of the clustering-based analysis engine.

In the experiments, each peer was set up as described in Section 5.4, and generated status update events, which were routed to the corresponding peer nodes using the content-based messaging substrate. Each node ran the cluster analysis algorithm. Networks of 100, 500, and 1000 peers were used, and each peer was set to generate 5 events during the data generation/collection phase. As in a real system, the value range for each attribute was normalized in order to conform to the dimensions of the information space. In this simulation, each attribute could take values between 0 and 255, and d was set with a value of 10.

The objective of the first set of experiments was to test for centralized (point) clusters, which occur when there are fixed configurations shared by all peers. In our example, certain types of peers can be recognized and characterized by specific attributes such as load, efficiency, etc. Ninetyfive percent of updates were generated around three fixed points — each update was a random perturbation of a given point, each of which had an approximately equal probability of being chosen. The remaining five percent of points were generated randomly. Figures 5.5a - d show the results for tests for different network sizes and cluster widths (produced by limiting the maximum perturbation of the central points). Anomalies are shown as circled points, while cluster centroids are shown as crosses.

The second set of experiments was similar to the first, except that instead of generating updates around fixed points, they were generated around a fixed line. This can occur when there is a fixed relationship between the monitored attributes, but the values of certain attributes, such as load, are highly variable. Again, a small percentage of updates was generated completely at random. Figures 5.6a - d show particular results for these tests.

The figures show how the distributed algorithm effectively identifies cluster and non-cluster points for clusters of different size, number, and shape. Notice that clusters

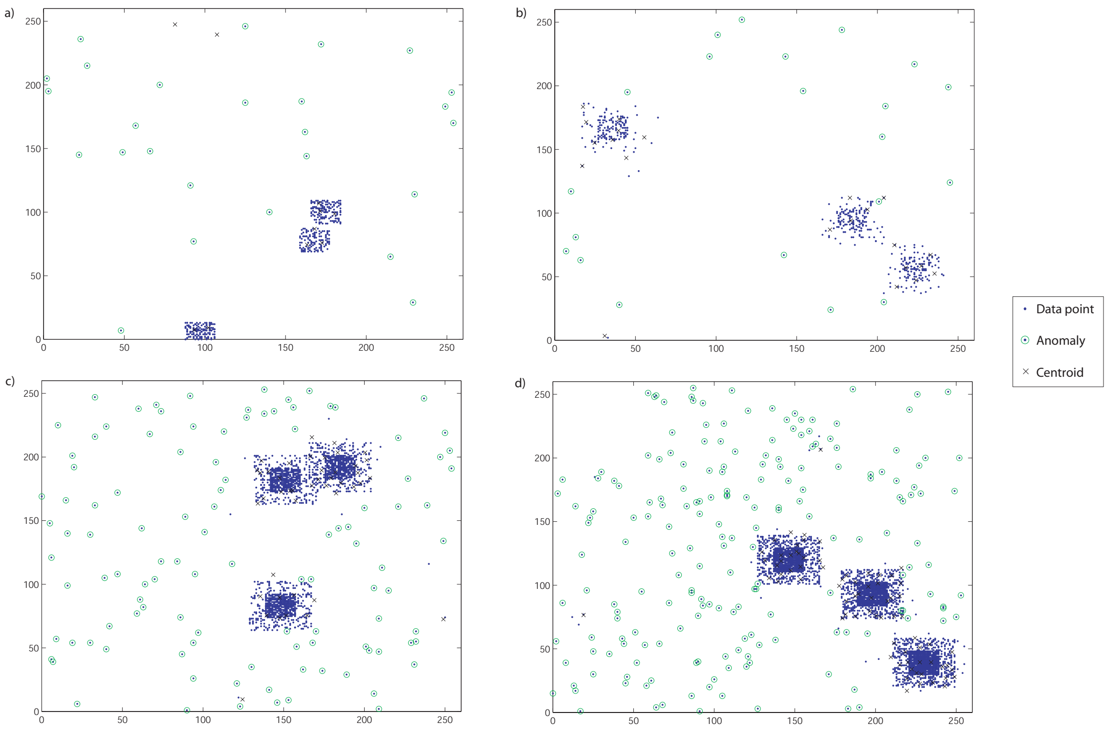

Figure 5.5: Analysis of point clusters for a) and b) 100 node network with different cluster sizes; c) 500 node network; and d) 1000 node network

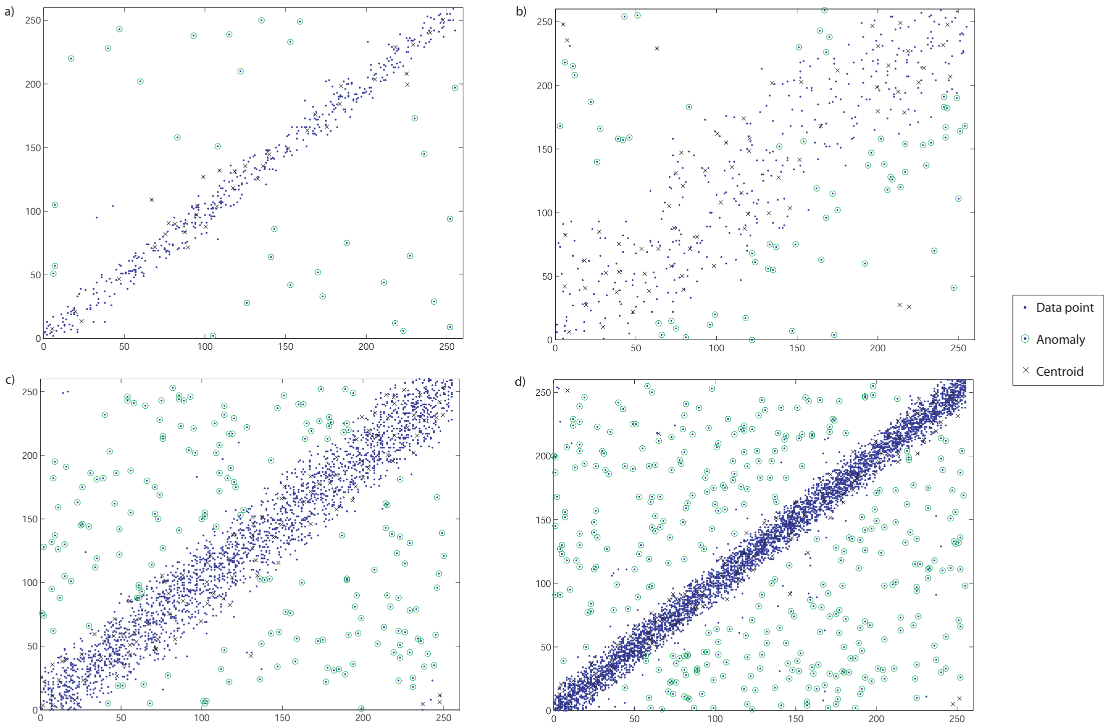

Figure 5.6: Analysis of line clusters for a) and b) 100 node network with different cluster widths; c) 500 node network; and d) 1000 node network

Network size	Point	Line
100	0.9%	0.4%
500	0.8%	1.6%
1000	1.3%	3.5%

Table 5.1: False positive rates for tests. The value is an average of the different tests done for different cluster widths (maximum perturbation).

are identified by differences in point density rather than by particular distances from cluster centers, so that different clustering behavior can be detected. To verify the accuracy of anomaly detection, we checked for points identified as anomalies when they were in fact close (within distance d) to an identified centroid. These false positive rates are listed in Table 5.1 as percentages of the total number of data points, and are due to boundary conditions. Although node communication deals with this problem as described in Section 5.3.2, it is possible that all neighbors may not be identified for a given region. This also explains why the rates increase with network size, since with more nodes the information space is divided into more regions, which means a greater number of region boundaries. However, the false positive rate is quite small and can be further reduced by increasing the communication between peers, but this will increase the cost of the algorithm.

An important observation is the relatively many centroids, shown as crosses in the figures, identified within each cluster. This is because each node with a region within the cluster will produce at least one centroid calculation. Thus, the number of centroids found is directly proportional to the number of nodes and inversely proportional to the distance d. In order to reduce this number and further consolidate notifications sent outside the network, the same neighbor communication used to deal with boundary conditions for anomalies can be used to aggregate cluster centroids from multiple nodes. Other aggregation techniques for peer-to-peer networks, such as those described in [29], can also be used.

Still, the cluster centroids represent a significant reduction of the information with respect to the raw data that can be more easily analyzed for trends and correlations between the different monitored attributes, without significantly reducing the accuracy of this analysis. In the experiments, the ratios of centroids to raw data points ranged from the order of 10:1 for the line cluster on the 100 node network to 160:1 for the point cluster on the 1000 node network. These reduction ratios will depend on the size of the network and the sizes of clusters present in the data.

As to performance, it is evident that the bulk of the complexity of the clusterin process is due to the mapping of data points to nodes which occurs prior to the analysis phase — the clustering itself is largely a local operation at each peer node. Each peer requires at most $O(n)$ messages, where n is the number of nodes in the network, and given that the Chord routing mechanism is $O(\log n)$, the messaging cost per peer node is $O(n \log n)$ [44]. Note that peer nodes exchange messages asynchronously. In contrast, the messaging cost of existing distributed implementations of k-means mentioned above depends on the number of synchronous iterations required to reach consensus among nodes for all centroids.

One tradeoff of this performance gain is load balancing. While other implementations distribute data points uniformly among nodes, in our design a small fraction of nodes, those responsible for regions that contain clusters, will handle a large fraction of data points. Our previous work has effectively addressed this load-balancing issue, basically by dividing single regions among multiple nodes [44]. However, the impact of these solutions on the self-monitoring applications has not been assessed and is an issue for future work.

Chapter 6

Conclusions

6.1 Thesis Summary

This research was motivated by two main factors: the importance of loosely-coupled communication in Grid systems and the emergence of Web Services for enabling standardized, platform-independent interactions between distributed resources in the growing infrastructure of distributed computing systems and particularly the Grid infrastructure. Recognizing that the Web Services Notification standard defines protocols for the publish/subscribe communication pattern, which is a suitable model for loosely-coupled communication, we presented a decentralized and content-based communication framework implementing this standard. This implementation was designed to address the main issues and challenges of large-scale and dynamic Grid systems.

Our approach leveraged the Meteor messaging framework, based on a rendezvous messaging model and content-based addressing scheme, and provided specific solutions to realize the interactions defined in the WSN standard, as well as optimizations for reducing the flow of messages exchanged within the system. To further support system decoupling and scalability, and reflecting the logical grouping of nodes that results from the interaction of virtual organizations in Grids, we designed a two-level overlay structure that interconnects groups of nodes without resorting to designated group heads that can become bottlenecks or single points of failure.

Finally, we explored the problem of in-network, decentralized data analysis for self-monitoring of peer-to-peer systems. We devised a mechanism for distributed clustering analysis that receives content-based status updates from monitored peers via our notification service and in turn generates notifications of anomalies and system trends. Our analysis approach takes advantage of the clustering properties of the mapping used by the routing mechanism of the communication infrastructure.

The experimental evaluations presented demonstrate the performance and scalability of the infrastructure as well as the effectiveness of the self-monitoring mechanism.

6.2 Observations and Future Work

While the functionality of the notification service has been analyzed and evaluated, the system is yet to be tested in a real application scenario. As described in [25], compatibility with other WSN implementations must be ascertained due to differences in the tools and development platforms used. Furthermore, real data and application contexts will test the flexibility and comprehensiveness of information space definition (e.g. in terms of dimensionality and resolution), for notification content descriptors as well as for monitoring attributes.

Our assumptions about the organizational divisions of computation resources that supported our two-level overlay design must also be challenged. Insofar as we extend our scope beyond the scenarios considered to a more global view of the Grid, such as those presented in [37, 18, 2], we must either consider further levels of hierarchy or generalize our high-level topology. Currently, all overlay groups are connected to each other and are identified using simple identifiers. It should be possible to extend this scheme by using content-based addressing at this level as well, and leverage content-based discovery services such as that which our framework can provide to determine the connections established between overlay groups.

The work presented here is part of an ongoing effort to develop a comprehensive set of middleware services for large-scale, dynamic distributed systems [37]. Besides the problems of communication and monitoring that have been addressed here, problems of information and resource discovery [44], task and workflow composition and coordination [32], and pervasive data aggregation [29] have and are being addressed by related work. The solutions that this and other work provide can be a basis for a generic workflow management and Grid application development platform.

Appendix A

Framework Implementation

This chapter presents some of the implementation details of the notification and monitoring framework, focusing on class organization and runtime procedures. The implementation of the framework is based on Java, though some of the components have been implemented in C++ and have not yet been ported to the Java platform. We first show, in Figure A.1, a high-level class diagram of the framework, identifying each architecture layer. The remaining sections give further details about each of the framework components.

A.1 Overlay

To define the overlay service, we first designed a generic interface that overlay implementations must realize. This gives the framework the flexibility to modify or change the overlay as necessary, though it does not completely make the framework independent of the overlay implementation. This is because, for the purpose of implementation, an overlay is considered to provide an addressing and routing scheme based on its particular topology. Users must be aware of the addressing scheme in order to effectively make use of the overlay. The `OverlayID` object provides an abstraction for this addressing scheme, but contains no specific functionality. All attributes and functionality are defined in its concrete subclasses, which are created for each overlay type.

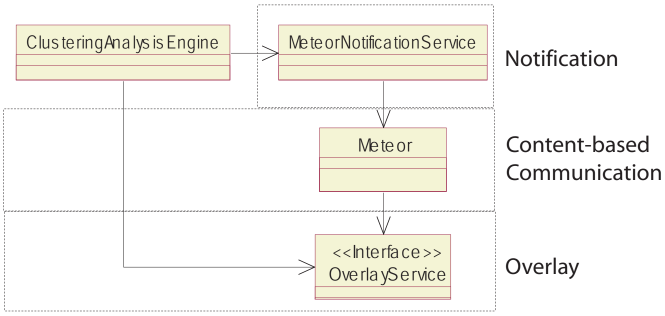
ClusteringAnalysisEngine
MeteorNotificationService
Notification
Meteor
Content-based Communication
<<Interface>> OverlayService
Overlay

Figure A.1: High-level framework class diagram corresponding to architectural design

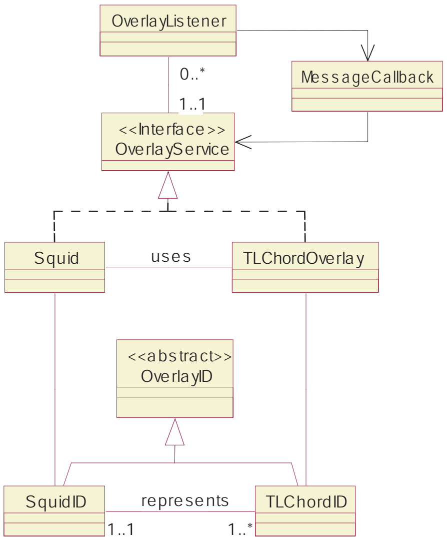

Figure A.2: Overlay layer class diagram

Notice that Squid is defined, along with the two-level Chord (`TLChord`) implementation, as a realization of the overlay service interface, because it is effectively provides its own addressing and routing scheme, relying to do so on the two-level Chord overlay. The `TLChordID` class encapsulates the m-bit Chord identifier, as well as a string that represents a group identifier, which were shown in Section 3.2.2 to be necessary to resolve an address in this overlay. The `SquidID` class in turn holds the sequence of key values that functions as a content-based address in Squid. As explained in Section 2.3.1, each Squid address will be mapped to one or more addresses from the Chord address space.

The functionality of the overlay service interface is defined as follows:

- `OverlayID join(url)`: Called when the node is initialized. The URL corresponds to the machine on which the node is running. Returns the overlay ID assigned to the node, or rather a concrete extension of the abstract `OverlayID` class that depends on the specific overlay's addressing scheme.

- `leave()`: Called when the node is terminated. Specific overlay implementations should handle these departures to maintain routing capability within the overlay.

- `routeTo(peers, tag, payload)`: Called to send a particular message to one or more nodes responsible for the given IDs. `peers` is a list of IDs, `tag` is an identifier for a listener of the overlay, and `payload` is the message to be sent.

- `url[] resolve(ID)`: Calls for the overlay to find the physical address of the node or nodes responsible for the given ID.
- `url[] getNeighborSet()`: An application may want to override an overlay's default routing scheme to implement different routing schemes (e.g. gossip) on the given overlay topology, for which the set of neighbors of a node is needed.
- `subscribe(listener, tag)`: Registers a listener to receive messages sent using the `routeTo` method to the given tag.

The implementation of the `join` method in the two-level Chord works in two ways. First, a node can be configured with a fixed address for a bootstrap node that can handle the join message. However, if such an address is not known, the joining node can broadcast the join message so that a node in the local network can act as the bootstrap node. If multiple nodes exist on the local network, a random backoff timer can be used to ensure that a single node acts as the bootstrap. If no node exists on the local network, the joining node assumes it is the first and completes the join operation accordingly. This mechanism replaces that of previous implementations of the overlay that relied on JXTA [30]. JXTA has the advantage that peers can be found even outside the local network, but it introduces overhead that can increase the latency of message exchanges (see Section 4.1.3; also see [3, 24] for a more thorough performance evaluation). Given the design of the two-level overlay, in which physical proximity plays an important role in group organization, the simple bootstrap mechanism is adequate.

Upper layers that make use of the overlay can receive messages sent through the overlay by subscribing as listeners with the `subscribe` method described above. A listener will also receive structural events for the given overlay (joining, failure, or departure of neighbor nodes), in case the higher-level service needs to be aware of and handle these changes (e.g. a DHT that needs to replicate the data stored on it at its neighboring nodes). Several different applications can share the overlay by subscribing with different tags, which are analogous to ports in the sockets API. The events received as a listener are handled by implementing the `OverlayListener` interface, which has the following methods:

- `messageArrived(MessageCallback)`: Called when a message is received destined for the listener.
- `serviceErrorOcurred(url, errorMsg)`: Called when a remote error corresponding to a message previously sent using the listener's tag is received.
- `leaving()`: Called after the `leave()` method for the overlay is invoked. Gives the listener a chance to perform tasks (e.g. offload data to another peer) when its node leaves the overlay.
- `newNeighbor(OverlayID, url)`: Called when a node joins and becomes a new neighbor of the listener's node. Gives the listener a chance to perform tasks (e.g. load balancing) to account for this new node.
- `neighborDown(OverlayID)`: Called when a neighbor node leaves or has been detected by the overlay to fail. Gives the listener a chance to perform tasks (e.g. repair and recovery) to account for the node absence.

The message callback object that is created and passed to listeners when messages arrive plays an important role in the design of the overlay API. In addition to the message that is being passed to the listener, the message callback object encapsulates the contact information of the source of the message. This allows it to provide a `reply` method, whereby receivers can reply directly to particular messages without a separate call to the overlay's `routeTo` method. The difference between the two is that the `routeTo` method may cause the message to be delivered in multiple hops as the overlay's routing protocol is invoked, whereas the callback's reply method sends the message directly through the underlying network.

More important than direct replies, however, is the ability to report back to the message source higher-level errors triggered by the reception of the message. Encapsulating error reporting along with the message that generated the error allows different

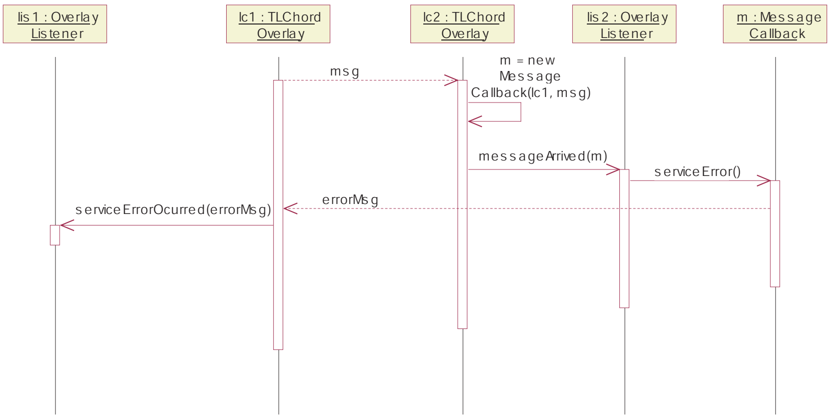

Figure A.3: Sequence for a message sent on the two-level overlay destined for a local node (lc2). The message is passed to a listener (lis2) within a MessageCallback object, which is used by the listener to report an error associated with the message. The error is sent back to the message source (lis1) through its corresponding overlay node (lc1). The dotted arrows represent asynchronous and possibly multihop messages.

overlay implementations to handle these errors in different ways, providing different implementations of the callback class. For example, recall from Section 3.2.2 that a message that is sent using the ANY group identifier over the two-level overlay is delivered first to a local destination and then is propagated further (to a corresponding node in a remote group) only if it cannot be handled on the local node. To keep implementation independence, only a higher-level service running on the overlay, such as a DHT, can determine whether or not a message can be handled. However, once an error is reported, the message callback object can determine the appropriate action. This behavior is illustrated in figures A.3 and A.4. Figure A.3 shows the default behavior of the callback object defined by the two-level overlay, in which the error message is directly returned to the source node after the error for a message is reported. However, when the ANY group identifier is used, the message is sent to a node in a remote group, which may be able to handle the message instead, as shown in Figure A.4. The error message is only sent back to the source if the message triggers errors in all groups, which is determined by the message callback implementation. Notice that, in order for this behavior to be possible, the error must be associated to the original message, which is why the callback object is used for error reporting instead of providing a method directly in the overlay service interface.

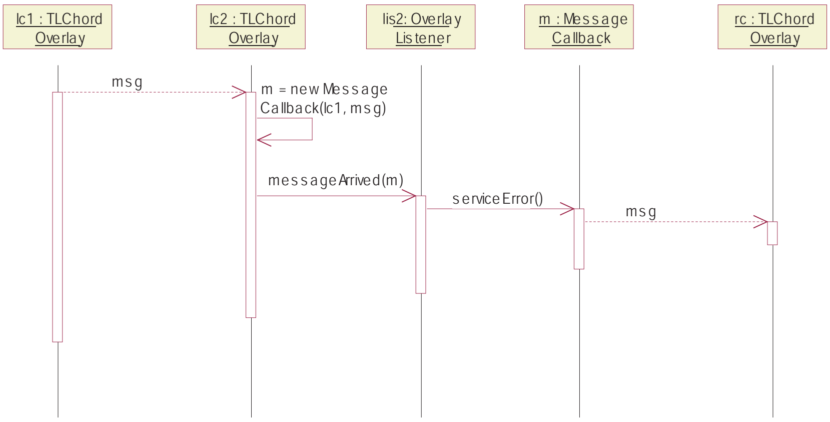

Figure A.4: Sequence for a message sent on the two-level overlay destined for any node and delivered first to a local node (lc2). The message is passed to a listener (lis2) using a MessageCallback object, which is used by the listener to report an error associated with the message. The original message is then resent to a node in a remote group (rc) that may be able to handle it. The dotted arrows represent asynchronous and possibly multihop messages.

A.2 Meteor Implementation

The implementation of the Meteor communication service consists of a single class that implements the `post` primitive described in Section 2.3.2. Each Meteor node stores messages with associated reactive behaviors and matches them based on the profiles of messages received. Figure A.5 shows the relationship between a meteor node and other system elements. Each node is responsible for messages with profiles that correspond to a particular region of the common information space, and only these messages are stored in the nodes local storage. The node relies on the mechanisms provided by the Squid overlay to route messages to the nodes that are responsible for them.

A.3 Notification Service

The Web Service interface for the notification broker was implemented in Java using the JWSDP 2.0 API and development tools. First, the XML schema for base notification and brokered notification, provided in [35], were transformed into Java objects using the JAXB binding tools, modifying some optional elements to conform to both the Java platform and our own implementation architecture, as follows:

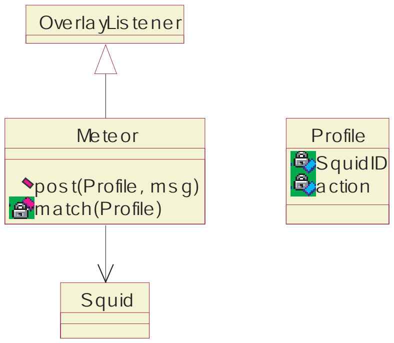

Figure A.5: Class diagram for the Meteor communication layer

- The FilterType element from the WSBN schema was redefined to contain a topic expression because topics are used for defining subscriptions.

- The TopicExpressionType element was redefined to contain a `xsd:string` element for holding topic strings as defined in WST.

- The Message element in the NotificationMessageHolderType definition was redefined to be of type `xsd:string`. This is not a limitation, since the message is application specific data that can be encoded as and interpreted from a string. This is mainly to facilitate its manipulation in Java.

- A SubscriptionReference element was added to the messages defined for subscription management (Renew, Unsubscribe, etc.), which is required to find particular subscriptions.

After the Java objects were created from the schema, the WSDL documents were used to create the Java service interfaces and implementing classes, which were then deployed as a service endpoint for a notification broker, shown in Figure A.6 as the `BrokerWSEndpoint` class. This endpoint can then be run on an Apache or similar web service container to receive and respond to client requests, which it transforms into method calls for the Meteor platform, extended by the `MeteorNotificationPeer` class to relay notifications and other WSN messages back to users through the endpoint interface. The current implementation is based on the JXTA deployment of Meteor, but can be easily extended to run on the two-level overlay implementation.

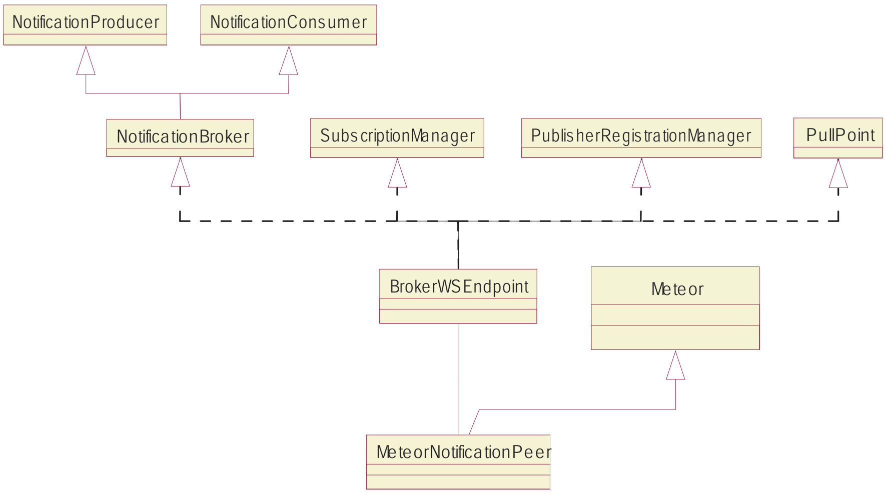

Figure A.6: Class diagram for the Notification Service

A.4 Monitoring Service

An implementation of the monitoring service must be split into two main parts. The first is an application-specific module that must run on every monitored node in order to obtain measurements of the monitored attributes and encode status updates according to a particular information space definition. The second is the clustering analysis engine, which receives these updates and runs the algorithm described in Section 5.3.2. The only implementation detail worth pointing out in this case is that the analysis engine cannot run directly on the notification service as any application that subscribes to notification events. Instead, it must intercept status updates directly from the overlay service (Squid, in this case), as shown in Figure A.1. This is because the analysis engine must receive all status updates received on the node, and, in order to do so as a subscriber to the notification service, would need to issue multiple subscriptions as the region assigned to the node changes with changes in the overlay. So, the analysis engine subscribes directly to Squid, at the same level as Meteor, but does publish its anomaly and centroid notifications via the notification service.

References

[1] Ioannis Aekaterinidis and Peter Triantafillou. Internet scale string attribute publish/subscribe data networks. In *Proceedings of the ACM 14th Conference on Information and Knowledge Management (CIKM)*, Bremen, Germany, October 2005.

[2] Mehmet S. Aktas, Geoffrey C. Fox, and Marlon Pierce. Fault tolerant high performance information services for dynamic collections of grid web services. *Future Generation Computer Systems*, 23(3):317–337, March 2007.

[3] G. Antoniu, P. Hatcher, M. Jan, and D. A. Noblet. Performance evaluation of jxta communication layers. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 1*, pages 251–258, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Apache Pubscribe project home: http://ws.apache.org/pubscribe/.

[5] Ozalp Babaoglu, Geoffrey Canright, Andreas Deutsch, Gianni A. Di Caro, Frederick Ducatelle, Luca M. Gambardella, Niloy Ganguly, Mark Jelasity, Roberto Montemanni, Alberto Montressor, and Tore Urnes. Design patterns from biology for distributed computing. *ACM Transactions on Autonomous and Adaptive Systems*, 1(1):26–66, September 2006.

[6] Roberto Baldoni, Carlo Marchetti, Antonino Virgillito, and Roman Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS '05)*, Columbus, OH, June 2005.

[7] Sanghamitra Bandyopadhyay, Chris Gianella, Ujjwal Maulik, Hillol Kargupta, Kun Liu, and Souptik Datta. Clustering distributed data streams in peer-to-peer environments. *Information Sciences*, 176(14):1952–1985, July 2006.

[8] Peter Bodik, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 89–100, Seattle, WA, June 2005.

[9] M. Castro, P. Druschel, A.M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication*, 20:1489–1499, 2002.

[10] M. Castro and A. Rowstron. Organizational locality in prefix-based structured peer-to-peer overlays. USPTO Patent Application No. 20040249970, 2004.

[11] Dave Chappell and Lily Liu. *Web Services Brokered Notification 1.3 (WS-BrokeredNotification).* Oasis, public review draft 01 edition, July 2005.

[12] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Proceedings of the First International Conference on Autonomic Computing*, pages 36–37, New York, NY, May 2004.

[13] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 23–32, Vienna, Austria, July 2002.

[14] Souptik Datta, Chris Giannella, and Hillol Kargupta. K-means clustering over a large, dynamic network. In *Proceedings of the Sixth SIAM International Conference on Data Mining*, Bethesda, MD, April 2006.

[15] Nicholas J. Harvey et al. System and method for creating improved overlay network with an efficient distributed data structure. USPTO Patent Application No. 20040054807, 2004.

[16] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114131, 2003.

[17] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–4, 2001.

[18] Geoffrey Fox, Sang Lim, Shrideep Pallickara, and Marlon Pierce. Message-based cellular peer-to-peer grids: Foundations for secure federation and autonomic services. *Future Generation Computer Systems*, 21(3):401–415, March 2005.

[19] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in g-major: Designing dhts with hierarchical structure. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 263–272, March 2004.

[20] L. Garces-Erice, E.W. Biersack, and P.A. Felber. Hierarchical peer-to-peer systems. *Parallel Processing Letters*, 13(4):643–657, December 2003.

[21] Steve Graham, David Hull, and Bryan Murray. *Web Services Base Notification 1.3 (WS-BaseNotification)*. Oasis, public review draft 01 edition, July 2005.

[22] GT4 tutorial: http://gdp.globus.org/gt4-tutorial/multiplehtml/index.html.

[23] Abhishel Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. *Lecture Notes in Computer Science*, 3231:254–273, 2004.

[24] Emir Halepovic and Ralph Deters. The jxta performance model and evaluation. *Future Generation Computer Systems*, 21(3):377–390, 2005.

[25] Marty Humphrey, Glenn Wasson, Jarek Gawor, Joe Bester, Sam Lang, Ian Foster, Stephen Pickles, Mark McKeown, Keith Jackson, Joshua Boverhof, Matt Rodriguez, and Sam Meder. State and events for web services: A comparison of five ws-resource framework and ws-notification implementations. In *14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, pages 24–27, Research Triangle Park, NC, July 2005.

[26] Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. Multi-resolution abnormal trace detection using varied-length n-grams and automata. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 111–122, Seattle, WA, June 2005.

[27] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. *Cluster Computing*, 9(4):385–399, 2006.

[28] Nanyan Jiang, Cristina Schmidt, Vincent Matossian, and Manish Parashar. Enabling applications in sensor-based pervasive environments. In *Basenets 2004*, San Jose, CA, October 2004.

[29] Nanyan Jiang, Cristina Schmidt, and Manish Parashar. A decentralized content-based aggregation service for pervasive environments. In *Proceedings of the International Conference of Pervasive Services (ICPS)*, June 2006.

[30] Jxta website. www.jxta.org.

[31] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *Proceedings of the First International Conference on Autonomic Computing*, pages 28–35, New York, NY, May 2004.

[32] Zhen Li and Manish Parashar. Comet: A scalable coordination space in decentralized distributed environments. In *Proceedings of the second International Workshop on Hot Topics in Peer-to-peer Systems (HOT-P2P)*, pages 104–111, San Diego, CA, July 2005. IEEE Computer Society Press.

[33] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing*, pages 84–95, New York, NY, June 2002.

[34] P. Niblett and S. Graham. Events and service-oriented architecture: The oasis web services notification specifications. *IBM Systems Journal*, 44(4):869–886, 2005.

[35] OASIS WSN Technical Committee: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.

[36] Shrideep Pallickara and Geoffrey Fox. Naradabrokering: A middleware framework and architecture for enabling durable peer-to-peer grids. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, pages 41–61, 2003.

[37] Manish Parashar and James C. Browne. Conceptual and implementation models for the grid. In *Proceedings of the IEEE*, volume 93, pages 653–668, March 2005.

[38] pyGridWare project homepage: http://dsd.lbl.gov/gtg/projects/pyGridWare/.

[39] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, San Diego, CA, 2001.

[40] Antony Rowstron and Peter Druschel. Pastry: Scalable, descentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[41] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[42] M. Sanchez, P. Garcia, J. Pujol, and A.F. Gomez. A novel design schema for hierarchical dhts. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, 2005.

[43] Cristina Schmidt. *Flexible Information Discovery with Guarantees in Descentralized Distributed Systems*. PhD thesis, Graduate Schoo - New Brunswick, Rutgers University, 2005.

[44] Cristina Schmidt and Manish Parashar. Flexible information discovery in descentralized distributed systems. In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12'03)*, 2003.

[45] M.T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The enterprise service bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797, 2005.

[46] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*, pages 73–86, Pittsburgh, PA, 2002.

[47] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, CA, August 2001.

[48] David Tam, Reza Azimi, and Hans-Arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. *Lecture Notes in Computer Science*, 2944:138–152, 2004.

[49] William Vambenepe. *Web Services Topics 1.3 (WS-Topics)*. Oasis, public review draft 01 edition, December 2005.

[50] WSRF.NET Project homepage: http://www.cs.virginia.edu/ gsw2c/wsrf.net.html.