DECENTRALIZED ONLINE CLUSTERING FOR SUPPORTING AUTONOMIC MANAGEMENT OF DISTRIBUTED SYSTEMS

BY ANDRES QUIROZ HERNANDEZ

A Dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2010

ABSTRACT OF THE DISSERTATION

Decentralized Online Clustering for Supporting Autonomic Management of Distributed Systems

by ANDRES QUIROZ HERNANDEZ

Dissertation Director: Professor Manish Parashar

Distributed computational infrastructures, as well as the applications and services that they support, are increasingly becoming an integral part of society and affecting every aspect of life. As a result, ensuring their efficient and robust operation is critical. However, the scale and overall complexity of these systems is growing at an alarming rate (current data centers contain tens to hundreds of thousands of computing and storage devices running complex applications), making the management of these systems extremely challenging and rapidly exceeding human capability.

The large quantities of distributed system data, in the form of user and component interaction and status events, contain meaningful information that can be used to infer the states of different components or of the system as a whole. Accurate and timely knowledge of these states is essential for verifying the correctness and efficiency of the operation of the system, as well as for discovering specific situations of interest, such as anomalies or faults, that require the application of appropriate management actions.

Autonomic systems/applications must therefore be able to effectively process the large amounts of distributed data and to characterize operational states in a robust, accurate and timely manner. Although highly accurate, centralized approaches for distributed system management are infeasible in general because of the costs of centralization in terms of infrastructure, fault tolerance, and responsiveness. Since data is

naturally distributed and the collective computing power of networked elements (ranging from sensor and device networks to supercomputer clusters and multi-organization grids) is enough to be harnessed for value added, system-level services, online and decentralized approaches for monitoring, data analysis, and self-management are not only feasible, but also quite attractive.

This work is based on the premise of realizing and applying online data analysis, exploiting the collective computing resources of distributed systems for supporting autonomic management capabilities. Specifically, we propose and develop decentralized online clustering as a data analysis mechanism and infrastructure, evaluate its accuracy and performance with respect to other known clustering methods, and apply it to the following autonomic management problems: 1) System profiling and outlier detection from distributed data, 2) definition, autonomic adaptation, and application of management policies, and 3) VM provisioning and energy management in data centers.

Acknowledgements

I would like to thank my advisor, Dr. Manish Parashar, for his direction, insight, and support, especially in granting me this invaluable opportunity in the first place. To Nathan Gnanasambandam, for his share of the work and the numerous suggestions that invariably made my work better. To my labmates throughout my stay at Rutgers, Cristina Schmidt, Nanyan Jiang, Ciprian Docan, Viraj Bhat, Vincent Matossian, and Ivan Rodero for all their help and camaraderie on and off the line of duty. And especially, to my parents, who may say that they don't always understand my work, but who will always be the first to praise it.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Distributed computational and information infrastructures, as well as the applications and services they support, are increasingly becoming an integral part of society and affecting every aspect of life. However, the scale and overall complexity of these systems is growing at an even more alarming rate (current data centers contain tens to hundreds of thousands of computing and storage devices running complex applications) and their management is rapidly exceeding human ability, making autonomic approaches essential.

An autonomic system must able to monitor and analyze system state and behavior and to discover patterns or anomalies. Additionally, these dynamically changing states must be evaluated against higher-level system goals or desired operation parameters, as expressed by automated policies or by human administrators, enabling the timely application of appropriate management actions. In order to realize this capability, autonomic systems/applications must effectively process the large amounts of monitoring data to characterize operational states in a robust, accurate and timely manner.

Although highly accurate, centralized approaches for distributed system management are infeasible in general because of the costs of centralization in terms of infrastructure, fault-tolerance, and responsiveness. Since in these systems data is naturally distributed and the collective computing power of networked elements (ranging from sensor and device networks to supercomputer clusters and multi-organization grids) is enough to be harnessed for value-added, system-level services, online and decentralized approaches for monitoring, data analysis, and self-management are not only feasible, but also quite attractive. This work is based on the premise of realizing online and decentralized data analysis, exploiting the collective computing resources of distributed systems, for supporting autonomic management capabilities.

1.2 Problem Description

In a distributed system, multiple sources produce events that represent current operational, functional, and control status, behavior, requirements, etc. These events can be produced by system components based on local information, or they can be the result of monitoring remote components or system users. Patterns that emerge in sequences of events are an important source of information about local and global system state, enabling the discovery of trends, anomalies, and/or violations that point to possible failures or other conditions of interest for the application of system management policies and control mechanisms. Often, these patterns can be recognized as clusters of events that can be interpreted as the normal behavior of groups of similar components, or of single components over time. Thus, analyzing monitoring events using clustering techniques can help to characterize system behavior, and also as importantly, to find potential deviations from normal behavior. The final goal is to be able to realize proactive management, which implies being able to predict possible future system states and situations of interest in order to respond to them to optimize efficiency and performance and to curtail undesired behavior.

Performing clustering analysis on monitoring data in a distributed environment and using the results of this analysis for autonomic management presents a number of challenges. Many existing self-monitoring systems, typically for web-based applications, perform data analysis offline and often in a centralized manner [45, 18, 36, 37], producing models that can then be used for runtime self-management. The derivation of these models often requires sophisticated processing and data mining techniques that can be expensive to run, especially if they must be run in addition to the regular workload of a system. Also, the types of behavior patterns that need to be detected are typically not known a-priori and cannot be directly encoded into the self-monitoring mechanism. In order to do online data analysis for autonomic management, it is important to implement the analysis in a decentralized and in-network fashion, using existing network resources and minimal extraneous information, which in turn ensures scalability, computational tractability, and acceptable response times.

A self-managing system must not only be able to recognize behavior patterns and classify events accordingly, it must also be able to characterize and properly interpret these patterns and relate them with corresponding control actions and management policies. Therefore, a clustering mechanism used for this purpose must provide mechanisms for cluster characterization and interpretation, in terms of management properties and attributes, which will result in dynamic profiles of system behavior. These profiles can then be used to identify types of behavior or operation, user roles, or system states to which management policies can be associated.

Finally, because self-monitoring mechanisms are subject to the same failures that occur in the network that they are helping to manage, the robustness of these mechanisms is of great importance to ensure overall system reliability at different levels. At the network level, the connectivity of the network must be maintained despite node failures. Next, at the data messaging level, the loss of the events that are important for the accuracy of clustering analysis results must be prevented. And finally, at the application level, specific knowledge can be provided to predict possible failures and recognize the most important data to enhance both the accuracy and efficiency of the robustness mechanisms provided at lower levels.

1.3 Problem Statement

There is a gap that exists between offline data analysis and predictive models for runtime system monitoring and management, and reactive online approaches based on empirical rules and predefined operating conditions and policies. The former rely on sophisticated data analysis techniques and are able to construct complex and accurate system models. However, this is often at the expense of accounting for changes in operating or usage conditions that require dynamic model reconfigurations because they incur in expensive and centralized computation. The latter, on the other hand, are meant to effectively handle dynamicity and keep the system within acceptable operating parameters, but are generally not suited for supporting proactive decisions because of limited knowledge available at the time of their design and the information that they can obtain from online data. While both approaches, or a combination of them, may be perfectly suited for the

management of a wide range of systems, we claim that decentralized and online data analysis (in particular, clustering analysis) can bridge the gap that exists between the two by extracting more information from online monitoring data in a timely manner, enabling more proactive decision making and management for a number of distributed applications, while taking advantage of decentralized system resources.

1.4 Research Overview

The research presented herein will develop and evaluate an autonomic distributed management framework based on the multidimensional monitoring and semantic online analysis of system events. These events are assumed to be in the form of periodic updates of behavior and operational status originating at system components and defined in terms of known attributes. The event attributes are used to construct a multidimensional space, which is then used to measure the similarity of events. Similar events corresponding to groups of components or component behavior over time can be identified by clusters formed in this space, while isolated events will identify components with abnormal behavior.

The basic approach for cluster detection consists of evaluating the relative density of data elements within the space. In order to evaluate data density, the multidimensional space is divided into regions, and the number of data elements within each region is observed by an individual processing node. If the total number of elements in the space is known, then a baseline density for a uniform distribution of elements can be calculated and used to estimate an expected number of data elements per region. Clusters are recognized within a region if the region has a relatively larger element count than this expected value. Conversely, if the count is smaller than expected, then these data elements may potentially be outliers. It is evident that the clustering algorithm itself requires minimal computation at processing nodes, which makes it suitable for online execution. We compare the proposed decentralized online clustering (DOC) algorithm to other known clustering algorithms in terms of accuracy and performance and present the result of this comparison along with conclusions supporting the conditions in which the use of DOC is advantageous.

The exchange of components' status updates is supported by a content-based messaging substrate [76], which also enables the partitioning of the multidimensional space into regions by implementing a dynamic mapping of data elements to processing nodes. The messaging substrate is responsible for getting the information used by the clustering analysis to the distributed processing nodes in a scalable fashion. The substrate essentially consists of a content-based Distributed Hashtable (DHT) that uses a Peano-Hilbert space-filling curve [75] as a locality preserving hash function. Content-based DHTs provide an interface that allows networked nodes to be addressed by attribute-value pairs. This means that data elements themselves can be used as addresses to send messages to particular nodes.

The robustness of the decentralized mechanisms is dealt with at three levels. First, the overlay network that supports the messaging substrate must guarantee the routing of messages and maintenance of the overlay structure despite membership changes and node failures. We leverage the self-organizing and self-healing mechanisms of overlay implementations such as Chord [81], which routes around failures by using alternate entries from its routing tables and periodically checks and updates the links in these tables. At the data messaging level, we use the concept of replication chains [67] to replicate and recover data elements to ensure their availability to the clustering algorithm. A replication chain is a series of successive nodes in the overlay network, with the property that if one fails, then the dynamic DHT mapping automatically maps to the next node in the sequence. This way, nodes automatically receive and recover elements from and for their neighbors in the chain. Finally, instead of having replication chains of a fixed length, we use the concept of selective replication to probabilistically replicate only those data elements that are likely to have an impact on the result of the clustering analysis, using Bayesian probabilities and historical data.

Finally, we also explore how to use clustering results for decision-making, as a way to close the loop of the autonomic management cycle. This is done via the dynamic application of management policies defined in terms of profiles constructed from online analysis. The mechanisms that are developed will be applied to different use cases, including device network and data center resource management and usage control.

1.4.1 Contributions

The contributions of this work are the following:

- A decentralized online clustering algorithm and its robust implementation based on a distributed monitoring and messaging infrastructure.
- Evaluation of the clustering algorithm and the tradeoffs and conditions for its application.
- A conceptual framework for the creation of system profiles from clustering results and the definition and dynamic application of management policies.
- Demonstration of applicability and results from actual use cases and system data.
- Application framework for the deployment of the self-monitoring, clustering, and dynamic policy generation for autonomic distributed system management.

1.5 Impact of the Research

The distributed computational infrastructures and applications targeted in this proposal are a growing component of societys IT infrastructure, affecting every aspect of life, including services related to health, banking, commerce, defense, education and entertainment. Automated management strategies to increase efficiencies and reduce these costs and impacts are urgently needed. This project aims to develop a monitoring and management approach that can effectively support these strategies. In addition to this, our work is guided by input and requirements given by industry partners through the Center for Autonomic Computing as part of the NSF I/UCRC program. We expect that the experience, products, and results of this research will contribute to the state-of-the-art in the development of products and services of these industry partners, and to the academic community in the fields of autonomic computing and distributed systems.

1.6 Dissertation Overview

The rest of this document is organized as follows. Chapter 2 discusses related work in the different areas touched on by this research, including autonomic management models and clustering algorithms. Chapter 3 then gives details on the architecture and robustness mechanisms used as the implementation infrastructure of our approach. Chapter 4 describes the decentralized online clustering (DOC) algorithm, discusses specific considerations for its application, and presents the results of a comparative analysis meant to demonstrate the accuracy and performance of the approach with respect to centralized clustering approaches. Finally, Chapter 6 summarizes the main conclusions of this research.

Chapter 2

Background: Autonomic Management and Data Analysis

There are two fundamental approaches for autonomic management that have been explored in the literature: top-down and bottom-up. The bottom-up approach focuses on the emergence of coordinated global behaviors based mainly on local interactions of relatively simple components, with little or no explicit direction or control. By contrast, the top-down approach considers explicit control elements that regulate the behavior of system components and that interact with each other, possibly through a control hierarchy, to produce the coordinated behavior that characterizes an autonomic system. Our research falls into the category of top-down approaches because it provides explicit mechanisms for monitoring and analysis of system data for the support of control policies and management actions.

The most general and established vision for top-down autonomic management is the MAPE control model [43]. We therefore describe the main aspects of this model and discuss different approaches for autonomic management that are based on it and that provide the context for our approach. As mentioned above, our main focus within the phases of the model is monitoring and data analysis (particularly, clustering analysis), as they support decision making, policies, and policy-based management.

2.1 MAPE: Autonomic Management Loop

The vision of Autonomic Computing, as proposed in [43], is of systems composed of elements that monitor their own behavior (and possibly that of other elements), analyze and evaluate this behavior against local and system goals, create appropriate plans of action intended to make local and global behavior match or approximate these goals, and execute these plans via available system actuators and component interactions. This view of autonomic components or elements is what is commonly referred to as the MAPE (Monitor, Analyze, Plan, Execute) control model, and is illustrated in

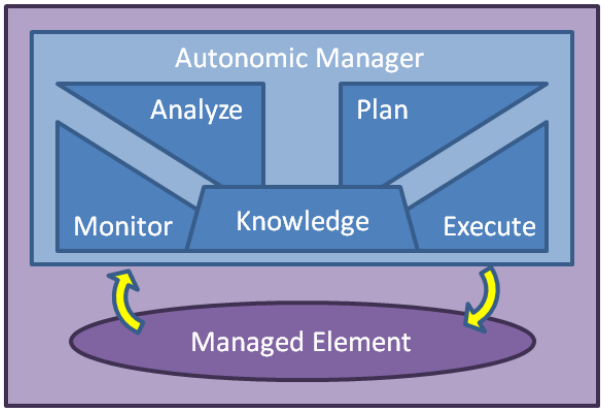

Figure 2.1: MAPE control loop

Figure 2.1.

Autonomic elements and autonomic systems are expected to exhibit certain autonomic behaviors or self-* behaviors [90], listed below:

- *Self-configuration*: Seamless integration of new components into a system, with automatic setting of operating parameters and discovery and establishment of communication between interacting components. Autonomic systems self-configure by means of a process called goal-driven self-assembly [19], in which elements are given a description of their task and the types of components that they require in order to complete it. Elements expose the services that they provide so that they can be discovered by those that require these services, and assemble in such a way that service requestors can have their requirements fulfilled by the appropriate service providers.

- *Self-optimization*: Continuous improvement of performance and efficiency via response to context, environment conditions, current interactions, etc. Because self-optimization at a system level is related to, but not necessarily guaranteed by self-optimization at the component level, special control policies or mechanisms of coordination and/or negotiation are required for achieving optimality at the system level.

- *Self-healing*: Automatic detection, diagnosis, and repair of localized software and

hardware problems, and system-level resilience to component failures or misbehavior. A self-healing component/system must be able to monitor its supporting/constituent components to determine if they are performing within their required parameters. Multiple component, system and application dependent mechanisms exist for recovering when components fail to perform as required, including proactive reconfiguration, replacement, load migration/redistribution, etc.

- *Self-protection*: Automatic defense against malicious attacks or cascading failures. Self-protection implies early (proactive) warning to anticipate and prevent systemwide failures. Security and usage control policies [62] are important elements for self-protection, since they define the parameters necessary to discover and associate the individual and, more importantly, correlated collective behaviors that precede these types of failures. Of course, these policies also specify the mechanisms to counter these behaviors, isolate malfunctioning components, and protect critical components and data.

In the following sections, we give details and describe existing work on the monitoring and analysis stages of the MAPE model, which are those that are relevant to this work.

2.1.1 Monitoring and Data Analysis

"Monitoring will be an essential feature of autonomic elements. Elements will continually monitor themselves to ensure that they are meeting their own objectives, and they will log this information to serve as the basis for adaptation, self-optimization, and reconfiguration. When coupled with event correlation and other forms of analysis, monitoring will be important in supporting problem determination and recovery when a fault is found or suspected." [43]

The distinction between monitoring and analysis in MAPE systems is sometimes fuzzy and there is no generally accepted definition of either. Monitoring is usually used to refer to the measurement and distribution/recording of status data, although it sometimes also encompasses some pre-determined processing of this data. This is

processing using functions, rules, or models that do not change based on the data. Data analysis, on the other hand, takes place when processing of the data is done in order to create or adapt models, rules, or descriptions of the system that can subsequently be used for monitoring and decision making.

Monitoring technologies, tools, and mechanisms

Monitoring depends both on the sensors (i.e. instrumentation) used to obtain status measurements from managed elements and on the media used to communicate the monitoring information to the elements that will process it. In autonomic computing literature, it is generally assumed that physical sensors and software instrumentation can be provided to obtain the necessary data. Media for monitoring includes log files, event-based messaging platforms, and query-based API's.

Event-based messaging systems, such as publish-subscribe systems, are common platforms for monitoring and are the ones assumed for this work. They typically ensure efficient and scalable delivery of messages in a loosely coupled system, such as that of autonomic components or agents. Common production publish-subscribe platforms include the Enterprise System Bus (ESB) architecture [57, 77]. Grid and wide-area network messaging and monitoring systems include the Globus Toolkit web service notification libraries [54, 31] and specialized grid monitoring tools for performance and availability [44, 58]. Also widely used are content-based publish-subscribe P2P networks. Distributed Hash Table (DHT) functionality is usually built using some sort of structured overlay network, the most popular of which are Chord [81], Pastry [74], and CAN [71], because they provide scalability, search guarantees and bounds on messaging within the network, as well as some degree of self-management and fault tolerance with respect to the addition/removal of nodes. Reliable message delivery in these systems has also been addressed [61]. In Chapter 3, we present the particular robust, content-based DHT on which our work is based.

Systems of autonomous agents are often used for monitoring and typically rely on a messaging infrastructure such as publish-subscribe [85]. Agents are software entities that run continuously and independently and are able to respond to and effect changes

in their environment [12]. Examples of autonomic systems that use agents for event aggregation and filtering, and some event correlation, are [7, 85]. In [65], more sophisticated analysis engines are plugged in as agents in a monitoring platform for intrusion detection.

One of the most critical aspects of monitoring, especially of large scale distributed systems, is dealing with the myriad components and large amounts of data efficiently and without excessive consumption of resources that could otherwise be used for functional tasks [43]. There are many alternatives in the literature that have been proposed to address this issue. Solutions have been proposed to keep track of the many different components and resources to be able to monitor them as they join, become active, and leave the system [24]. Because these registration-based solutions have limited scalability and flexibility, decoupled, content-based communication, such as that offered by the publish-subscribe systems cited above, is often used so that keeping explicit track of components is not necessary.

With respect to data management, one option is to use monitoring logs as buffers that can be analyzed offline without putting too much burden on the system. For example, [64, 33] propose technique for log analysis and summarization, and many of the analysis approaches mentioned in the next section rely on offline data analysis from application logs. Applying statistic sampling techniques is also a viable option for reducing the amount of monitoring data. Well known design of experiment [10] sampling techniques, including random sampling [13] are useful for systems with unknown configurations or data distributions. More sophisticated sampling and modeling techniques like probabilistic collocation [89, 87, 50], in which the importance or interest distribution of data is updated over time and used to sample more interesting/important data with higher probability. Other approaches for selective monitoring include [94], which suppresses monitoring updates from distributed nodes based on the correlation of values from individual nodes, as well as from groups of nodes.

Data Analysis and Modeling

With such a broad topic, in this section we can only hope to briefly mention some of the analysis and modeling approaches, focusing on their use in existing autonomic management solutions. Regardless of the method used, the basic approach is to use data analysis to build a full or partial model of the monitored component or system, which characterizes normal system behavior and is then used to drive management actions (detect anomalies, predict system response, etc.).

One of the most common forms of data analysis and model types are statistical regression and data correlation. Simple or domain-specific correlations and linear regressive models can be constructed and applied to online data. Examples of this are common in intrusion detection systems [63] and workload prediction [93]. Auto-regressive and moving average models like Box-Jenkins ARIMA or ARMAX [11] are also used for workload modeling and prediction [91, 17]. Kusic et al. [48] proposed a lookahead control algorithm and used it to identify optimal allocations of computing resources to multiple client QoS classes under correlated job arrivals modeled by ARIMA processes and service demand given by an exponentially weighted moving average filter. Other related models like quadratic response surface models (QRSM) [56, 68] are useful when the data varies significantly in terms of long-range correlations in the signals and non-linearity. Unlike linear regressive models, these models require larger data sample sizes.

An important application of correlation models can be found in [37], for normal application component interactions based on user requests, by measuring the intensities of data flows between components and modeling invariant relationships between input and output flows at each component. In [38], real-time system monitoring data is used to adaptively trigger sets of rules for generating alerts (which are the result of management policies). The approach described uses system invariants (static functional relationships between data flows of dependent attributes) to correlate multiple measured attributes and determine the probability of false positives in the triggering of management rules. This is done by determining if the pattern of triggered rules indeed corresponds to the

invariant relationships.

Queuing and time-series models have been heavily employed in web-server and data center management. Their application requires modeling component service and arrival times and the input-output dependencies of these components, which are found empirically or using simulations with workload traces or logs. Chandra et al. [17] modeled resources using a time domain queueing model to capture the transient behavior of application workloads, and used prediction algorithms to estimate future application workload parameters based on online measurements. Urgaonkar et al. [86] describe a dynamic provisioning approach for multi-tier Internet applications that considers the dynamic assignment of a number of servers to the different application tiers, based on short and long term workload characteristics.

Finite state automata are also constructed and used for input classification and anomaly detection. Automata constructed from normal user request traces in web applications are used in [36] to recognize abnormal interactions of application components that are the result of or lead to faults. Process Query Systems [20, 72] are used for recognizing known behavior patterns or processes, for which a state model is known, by separating out interwoven event streams. These events are assumed to correspond to the processes' event transitions, so that applying the state models in parallel as competing hypotheses, the generating processes can be discovered. Also used for a similar purpose are Markov chains [70] and neural networks [78]. All of these methods require training with labeled (pre-classified) data.

2.1.2 Online Data Analysis Approach

One thing to note about the majority of the analysis methods of the previous section is that they depend on offline training to develop the respective models used for runtime management. As stated by our problem statement, there is a gap that exists between offline data analysis and predictive models for runtime system monitoring and management, and reactive online approaches based on empirical rules and predefined operating conditions and policies. In order to account for changes in operating or usage conditions

that require dynamic model reconfigurations and still be able to effectively handle dynamicity and keep the system within acceptable operating parameters, solutions need to extract as much information as possible from online monitoring data in a timely manner. This way, enough information is available for proactive decision making, and it can evolve dynamically with the evolution of the system.

Because of its criticality and near real-time requirements, network security and specifically intrusion detection are areas where online data analysis plays a crucial role. Many of the issues that we have identified as motivations of this work have come up in the context of intrusion detection. The authors of [7, 47] identify the limitations of centralized analysis and attempt to reduce the data sent to a centralized node by using hierarchical event distribution in an agent-based framework. The probabilistic approach for anomaly detection proposed in [15] also has similarities with the present work, in that it examines events in time windows and looks for patterns. In the case of [15], the patterns are statistical autocorrelations in the data for the different dimensions being examined. Outliers are detected by values that differ by statistical average by a significant number of standard deviations. Our approach is similar in the search of patterns, but it applies a multidimensional and unsupervised analysis to the data (see Section 2.2). Gossip based intrusion and attack detection [29, 92] is also an important application of online data analysis. It is a very efficient analysis mechanism because every node acts solely on local information and information from its nearest neighbors. Given the right topology, information about an attack can propagate very quickly and efficiently to support well-informed decisions.

Clustering has also been applied to the problem of intrusion detection. Both [66, 40] use clustering to analyze offline data to train an online anomaly detector and to classify intrusion alarm types, respectively. In [88] a fuzzy clustering algorithm based on the concept of nearest neighbor connectivity is applied to intrusion detection. The complexity of this algorithm is in the same order as that of DBSCAN, introduced in Section 2.2.3 and analyzed in Section 4.2. Similarly, y-means clustering, which is based on k-means, was designed specifically for intrusion detection. It overcomes some of the drawbacks of k-means like its dependence on the value k, but like k-means it cannot

generally detect clusters of arbitrary shape.

2.1.3 Policies and Policy-based Management

Within the conceptual framework of the MAPE control model, policies are representations, in a standard external form, of desired behaviors or constraints on behavior [90]. Policies can be considered to be incorporated into the knowledge element of autonomic managers (see Figure 2.1) and affect (and can be affected by) the different stages of the control loop. Specifically, they enable an autonomic element or system to translate high-level, broadly scoped directives into specific actions. In a seminal paper on policy-based management for distributed systems, Sloman [79] described policies as pieces of information that influence the behavior of objects within a system, and, from a design point of view, are separated from managers that enforce them so that they can be changed without changing general enforcement mechanisms.

Although there is no single classification that distinguishes different types of policies, it is generally recognized that policies are expressed at different levels. According to [90], these levels are:

- *Action policies*: Policies that are usually expressed as conditional rules associated with specific threshold parameters and triggering specific low-level operations.
- *Goal policies*: Policies that describe conditions to be attained without specifying how to attain them. They allow giving direction to an element without requiring knowledge of the element's inner workings.
- *Utility policies*: Policies that specify the relative desirability of alternative states, automatically determining the most valuable goal in any given state.

One of the most important issues in policy-based management is policy refinement, from utility and goal policies to low-level action policies that can be applied to individual system components based on system state. In [53], the concept of policy hierarchies was introduced to formalize the kinds of refinement among different types of policies. At the time, it was stated that no automatic method for goal refinement (such as going

from “Protect from data loss” to “Backup weekly”) was possible because it depended on the real-world judgment of the manager. However, with the emergence of semantic ontologies and knowledge-based reasoning, as in [46], such automatic refinement can be realized, at least within specific contexts. In [82], Strassner describes a general conceptual model for managing context for policy-based systems, so that appropriate management policies at different levels can be selected.

Context is described as a collection of measured or inferred knowledge that describes system state or environment. However, aside from policy selection, dynamic context should also determine specific policy parameters, which are usually expressed as static thresholds or constraints on the policies' subjects. In fact, a recently proposed research agenda for distributed policy management [21] identified the re-computation of action policies (constraint thresholds) in reaction to changes in system state as a problem that is yet to be satisfactorily addressed.

The use of utility and utility functions for autonomic management, as in [42] and [23], is, in fact, complementary to work on policies, since utility functions can implicitly be considered to define utility policies. In other words, the attributes that constitute the space within which high-level policies are defined is directly related to utility (making the constraints set by policies at this level more stable). Thus, techniques for utility function management and elicitation can directly apply to policies in this space.

2.1.4 This Work within the MAPE model

The decentralized online clustering and dynamic policy framework that is the product of this research is not in itself a fully autonomic system, although it contains individual elements that are autonomic. For example, the messaging infrastructure described in Section 3.1 exhibits the autonomic properties of self-configuration and self-healing. Additionally, the robustness mechanisms designed for the clustering system and described in Section 3.2, provide self-protection properties.

The framework itself can be thought of as the monitoring and analysis components of a distributed autonomic manager of elements that are themselves distributed. The

distributed elements being managed may themselves be autonomic elements. In fact, we assume that the distributed elements are producing monitoring data, either by means of an autonomic manager or by way of an external monitoring agent (whichever is the case, is not relevant).

Note that the execution stage of the MAPE loop is supported to some extent by the messaging infrastructure, which matches events to policies to apply the actions that they define (see Section 5.1). However, the planning stage is not explicitly supported by our work and depends on pre-defined plans implicit in the definition of policies. High-level autonomic planning is outside the scope of our work.

2.2 Clustering Analysis

Clustering is the unsupervised classification of data elements (observations, data items, or feature vectors) into groups (clusters) [35, 34]. In general, clusters are identified based on some definition of similarity, such that the elements in a cluster are most similar to each other, whereas elements in different clusters are not. Unsupervised classification means that group labels or descriptors are not provided beforehand, so that they must be discovered solely from the input data.

Different approaches to clustering usually share the following basic steps [34]: 1) Element representation; 2) Definition of a element proximity (similarity) measure; 3) Clustering or grouping; 4) data abstraction (if needed); and 5) assessment of output (if needed). Steps (1) and (5) are mainly application dependent and will not be discussed here, although there are structural and statistical approaches for (5) that can be found in [35]. For step (4) clustering algorithms commonly use centroids or other representative points to describe clusters. Those abstractions used in this work are described in Section 4.1.1. The following subsection will enumerate different classes of clustering based on (2) and (3), and relevant work for those that are most related to this research.

2.2.1 Proximity Measures

Distance Measures

A distance measure (a specialization of a proximity measure) is a metric (or quasi-metric) on the feature space used to quantify the similarity of data elements, where similarity is inversely proportional to distance. Common distance measures include:

- *Euclidean or generalized (Minkowski) distance*: The Euclidean distance of two data element vectors x_i and x_j with d dimensions (attributes) is given by:

$$d_2(x_i, x_j) = \left(\sum_{k=1}^{d} (x_{ik} - x_{jk})^2 \right)^{1/2}$$

  which is a special case of the generalized distance:

$$d_p(x_i, x_j) = \left(\sum_{k=1}^{d} |x_{ik} - x_{jk}|^p \right)^{1/p}$$

  This distance is useful for data sets with compact or isolated clusters [52], but differences in scale among dimensions can affect its results, in a way that the largest scale attribute tends to dominate the others. This problem is usually handled by normalization of the data.

- *Mahalanobis distance*: Given by:

$$d_M(x_i, x_j) = (x_i - x_j) C^{-1} (x_i - x_j)^T$$

  where C is the covarience matrix of the data elements. It is useful when the data has linear correlations that can affect the use of other distance metrics.

- *String distances*: Used in syntactic clustering and include Hamming distance, edit distance, and other elaborate document comparison distances based on common terms [6]

- *Graph distances*: Clustering on graphs is usually done by assessing the similarity of data elements based on some notion of connectedness, either by measuring the number of links (edges) between groups of nodes or by measuring the distance in the length of paths between nodes.

Density Measures

Density measures can be considered a special type of distance measures that take into account the effect of surrounding or neighboring data elements on distance. Density measures are usually calculated taking into account the number of elements in a unit space, and since they are not necessarily calculated as pairwise distances, they are employed by different clustering algorithms than those defined for distance metrics. However, some distance metrics exist that take density into account, such as the mutual neighbor distance (mnd) [30]:

$$mnd(x_i, x_j) = nn(x_i, x_j) + nn(x_j, x_i)$$

where $nn(a, b)$ is the neighbor number of b with respect to a (i.e. the index of b in the nearest neighbor list of a). This definition effectively computes distances with respect to density because an element in a dense cluster has many more nearest neighbors and is thus farther away from an arbitrary element than an element outside a cluster. The nearest neighbor distances are expensive to compute, however, and are not commonly used for density-based clustering.

Density-based clustering algorithms have some advantanges with respect to those that are distance-based, especially when it is necessary to identify outliers (singular data elements or noise) and clusters of arbitrary size and shape. This is because distance-based algorithms usually produce a complete partition of the data into convex sets. Furthermore, these algorithms cannot identify clusters of arbitrary (non-convex) shape [26].

Other measures

Particular applications may have specialized proximity measures that exploit domain information. In general, similarity can be defined as:

$$d(x_i, x_j) = f(x_i, x_j, \mathscr{C})$$

where $\mathscr{C}$ is a set of domain-specific concepts. For example, in Figure 2.2, the distance

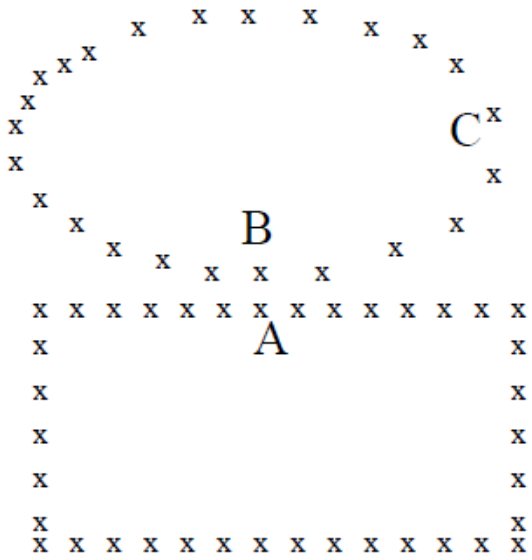

Figure 2.2: Clustering example that requires domain-specific concepts to define distance (Figure from [34])

between points A and B should be greater than the distance between points B and C, given the concepts of the shapes to which the points belong.

2.2.2 Clustering Approaches

Given a similarity measure, clustering techniques can be classified according to the following criteria:

- *Partitions*: A clustering approach is *hierarchical* if it produces a nested series of partitions, and *partitional* if it produces only one. A hierarchical approach is usually represented by a dendogram (see Figure 2.3), which is a tree with the root representing a single cluster of all patterns, the leaves representing each individual data element, and each level showing an intermediate partition of the data elements into clusters according to their distribution. The clustering algorithms considered in this work are all partitional.

- *Sequence*: A clustering approach is *agglomerative* if it begins with each data element in a single cluster and successively merges clusters until a stopping criterion is reached. It is *divisive* if it begins with a single cluster of all data elements and

then divides existing clusters. Agglomerative approaches are considered in this work.

- *Features used*: A clustering approach is *monothetic* if it partitions data elements based on individual features (dimensions) separately, and it is *polythetic* if it makes use of them all simultaneously, as in the computation of a pairwise distance. The approaches considered in this work are polythetic.

- *Cluster membership*: A clustering approach is *hard* or *exclusive* if it classifies every data element with a single cluster label, and it is *fuzzy* if it gives each data element a degree of membership in each cluster. Hard clustering is a special case of fuzzy clustering, where the membership equals zero for each data element for all but one cluster label. The approaches considered in this work are hard, although we consider a metric from fuzzy clustering in the comparative analysis of clustering algorithms of Section 4.2.

- *Progression*: A clustering approach is *incremental* if the results of clustering can be updated as new data elements are considered without examining previous elements. In a non-incremental approach, a non-empty subset of previous data elements must be reexamined in order to produce new results. Please see the algorithm comparison in Section 4.1.2 for a discussion on the differences in terms of the progression of the algorithms considered herein.

2.2.3 Clustering Algorithms

Because of the nature and requirements of self monitoring, this work deals with partitional, agglomerative, and exclusive clustering. This classification includes both distance and density based algorithms. Among distance-based algorithms, probably the most widely employed is k-means clustering. Traditional k-means is an iterative process that calculates the coordinates of k centroids (elements with minimum average distance to the data elements in their cluster) for a number of data elements. Both k and an initial estimate for the centroids must be given as input, and the algorithm successively

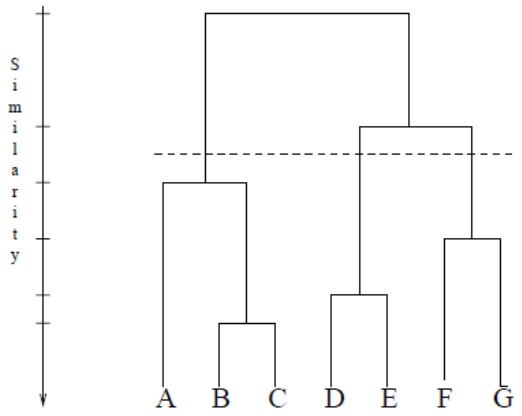

Figure 2.3: Example of a dendogram resulting from a hierarchical clustering (Figure from [34])

associates data elements to their nearest centroid and recalculates centroids until they do not change significantly from one iteration to the next.

A basic algorithm for density-based clustering is presented in [35]. These algortihms identify clusters based on comparing the densities (number of elements per unit space) around the elements in the data set. We now describe in detail two density-based algorithms that are used for comparison with our proposed algorithm in Section 4.2.

DBSCAN

DBSCAN [26] was designed for data-mining in large databases and uses the concepts of density-reachability and of core and border points to very effectively identify clusters of arbitrary size and shape. Basically, the DBSCAN algorithm takes two parameters: a distance value, called *EPS*, which determines a region around a point, (data element) called an *EPS-neighborhood*, and an integer value *MinPoints* that indicates a minimum number of points per region. Together, these two parameters define a density threshold to identify the points that belong to a cluster. A point that contains at least *MinPoints* within its *EPS-neighborhood* is called a core point and is automatically assumed to be part of a cluster. A border point is a point that does not satisfy the density threshold (core point condition), but that is within the *EPS-neighborhood* of a core point. A cluster is a maximal set of points obtained by finding all points that are density-reachable

from a starting core point (i.e. that are within the *EPS-neighborhood* of that point or within that of a core point that is density-reachable from it). Any point that cannot be found to be in a cluster in this way is considered noise. Although there are more recent algorithms that improve on DBSCAN for particular types of data, such as 2D data [32], or data distributions [41], DBSCAN continues to be one of the most prevalent general-purpose density-based clustering algorithms in the literature, making it well-suited as a benchmark for accuracy.

Clique

The Clique [4] algorithm divides a multidimensional space into regions of a given size and determines if each region (i.e. the data elements contained therein) is part of a cluster. This is the case if the percentage of elements within the region relative to the total number of elements being clustered is greater than a given minimum proportion of elements for each region (density threshold). A cluster is a maximal set of adjacent regions that exceed the density threshold.

As described in [4], Clique can be considered a monothetic algorithm, because it clusters on different dimensions separately to find the subset of dimensions that yields the best clustering result. However, in this work, we ignore this feature of the Clique algorithm and focus solely on the clustering principle explained above.

2.2.4 Distributed Clustering

Distributed algorithms for k-means have been devised [8, 22]. These procedures partition data elements uniformly among processing nodes, each of which runs the k-means procedure on their subset of elements. Nodes then repeatedly exchange centroids with neighbors and rerun the algorithm until a consensus is reached. The rate of convergence, which determines the number of iterations and thus the execution time of the analysis, depends on several factors, including the number of nodes and the actual distribution of the data. If the number of iterations required is large, then the process will be slow, considering online analysis.

An application of DBSCAN for a P2P system is presented in [49]. This is similar to

the present work in that it divides data among processing nodes locally, so that nodes process data elements in subregions of the multidimensional space. However, there every node runs the DBSCAN algorithm locally, which is more expensive computationally than the approach presented here. Our approach is based mostly on the Clique method, as presented in [4], and is, to our knowledge, the only distributed application of this kind of clustering algorithm.

The work in [59] shows an application of clustering in a distributed system, namely a large multi-agent system, which is based on measuring graph connectivity. In this type of system, agents need to form groups with other agents of similar characteristics. Agents are assumed to be connected initially to an arbitrary small number of other agents, and clustering is done by successively creating links with better matches provided by the views of agents' neighbors and dropping links between poor matches. The mechanism is very suitable for a multi-agent environment in which there is no pre-existing organization between agents. In contrast, in this work, we assume an existing system or virtual organization in which nodes are connected in a well-known, structured (albeit dynamic) way, and exploit this organization to transfer data between nodes.

2.2.5 Stream Clustering

There is also work on the problem of online clustering, dealing with clustering data streams, which is the case for monitoring. This work deals mostly with the evolution of cluster information and the summarization of cluster data over time. Because of this, it can be considered as an orthogonal and complementary issue to the actual clustering algorithm employed, although existing approaches are often tied to a particular clustering algorithm.

In the area of machine vision, the work in [2] explores the topic of stream clustering, with the purpose of tracking the movement of a known set of clusters (perceived objects) as they move in the field of vision. This work is limited because it does not consider the addition/removal of clusters over time. The work in [3] describes a framework for clustering evolving data streams in general, based on the concept of microclusters, which are evolving snapshots of the data stream found using large numbers of k in runs of the

k-means algorithm. To obtain macro results for specific time windows, the microclusters are merged using the standard k-means for a given k, which, like regular k-means, limits the approach to instances where k is known. In [16], the microclustering-based approach is extended by replacing the use of k-means with a variation of DBSCAN. Because of the complexity of DBSCAN, only the identification of microclusters is done online. The final clustering result is found using an offline process that merges the microclusters.

The windowing approaches used by the preceding projects, such as keeping data elements based on a weight that decreases over time, can be exploited for our framework for profile maintenance and characterization. Also applicable are the approaches described in [9], which is a survey on the temporal evolution of patterns in a generic sense. The approaches described therein are based on incremental mining and on the detection of interesting changes using learning.

Chapter 3

Infrastructure Support for Data Distribution

This chapter gives details about the different technologies and implementation mechanisms that support the autonomic data analysis and management framework developed in the rest of this document. First, we describe the content-based P2P communication platform that will be used to efficiently distribute events to the appropriate nodes for analysis (Chapter 4) and policy application (Chapter 5). Next, we also describe and analyze platform and application level mechanisms to ensure the robustness of the proposed framework to random and malicious failures and data loss.

3.1 The Meteor Framework

Meteor [39] is a content-based communication platform for peer-to-peer systems based on a rendezvous messaging model [80]. 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 3.1 shows the framework architecture and each of the component technologies, described below.

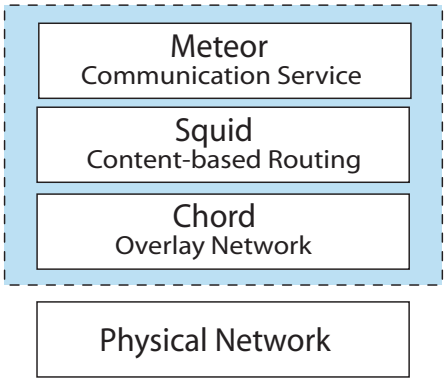

Figure 3.1: The Meteor framework

3.1.1 Squid: Content-based Routing Engine

Squid [76] implements a dynamic mapping between a multidimensional attribute space and a dynamic set of nodes. It is built as a distributed system on top of the indexed peers, which are organized using the Chord [81] overlay network. Its key innovation is a dimension reducing indexing scheme and routing mechanism that effectively maps descriptors in a multidimensional attribute space to physical nodes in the overlay network. Squid dynamically divides the multidimensional space among the distributed nodes, so that disjoint continuous subspaces of the multidimensional space are assigned to each participating node. Squid guarantees that all existing nodes to which a given descriptor 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 data points to the overlay's one-dimensional index space, so that points that are close in the multidimensional space will be close in the index space. This enables Squid to efficiently handle complex content-based descriptors with partial keywords, wildcards, and ranges, as these queries map to a reduced number of nodes.

The mapping of attribute descriptors to the index space, shown in Figure 3.2 is accomplished by using a locality preserving mapping called a Hilbert Space Filling Curve (SFC) [75]. 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 attribute space used to index some computation resource with regard to bandwidth and storage (Figure 3.2 (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 3.2 (b)). The dimension of the SFC maps to Chord's m-bit index space, so that this point corresponds to a Chord index (Figure 3.2 (c)).

Nodes in the Chord overlay have identifiers in this same index space, assigned to them randomly upon joining, and each node is responsible for the indices between its predecessor (the node with the immediately preceding id) and itself. Chord provides a

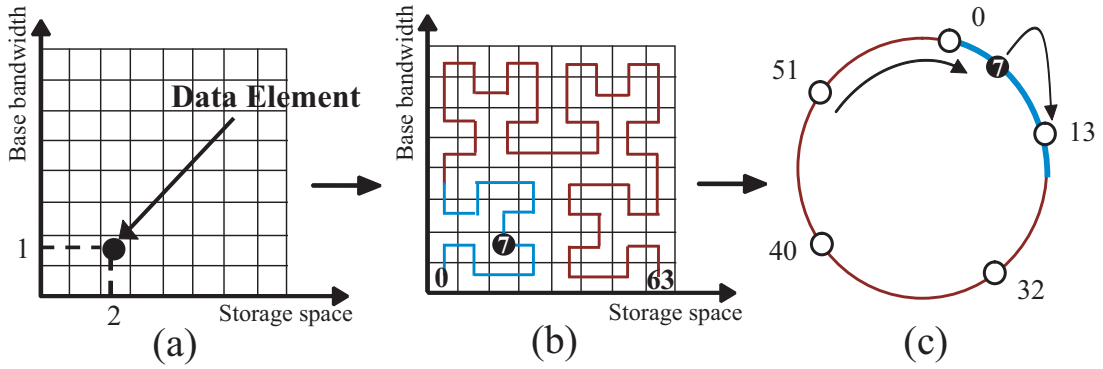

Figure 3.2: Mapping from a multidimensional attribute space to a node in the overlay network

lookup operation by which the node responsible for a given index (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 points that are close (in this case, lexicographically close) in the multidimensional attribute space will likely be mapped to indices that are local in the one-dimensional index space. Because of this, close indices will likely be mapped to nodes that are close in the overlay, or even to the same node. Figure 3.3 illustrates the search process using complex queries (e.g. ranges and wildcards). As Figure 3.3 (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 index resolution at multiple nodes in the system, ideally the nodes to which the complex descriptor 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 [76].

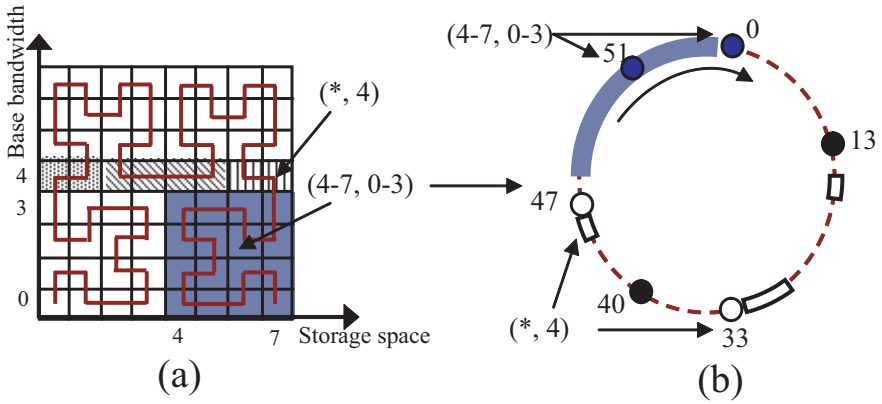

Figure 3.3: Searching the system: (a) regions in a 2-dimensional space defined by the queries (*, 4) and (4-7, 0-3); (b) the clusters defined by query (*, 4) are stored at nodes 33 and 47, and the cluster defined by the range query (4-7, 0-3) is stored at nodes 51 and 0.

3.1.2 Associative Rendezvous Communication Service

The Meteor communication service is a kind of DHT based on associative rendezvous, a paradigm for content-based decoupled interactions [39]. The nodes that implement the service function act as rendezvous points (RPs), which, like in typical rendezvous-based communication [27], 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 node, 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 node runs on top of a Squid node, and, correspondingly, is responsible for a portion of the multidimensional attribute space from which profiles are drawn. The Meteor node uses Squid's routing engine to deliver a message to corresponding RPs based on the

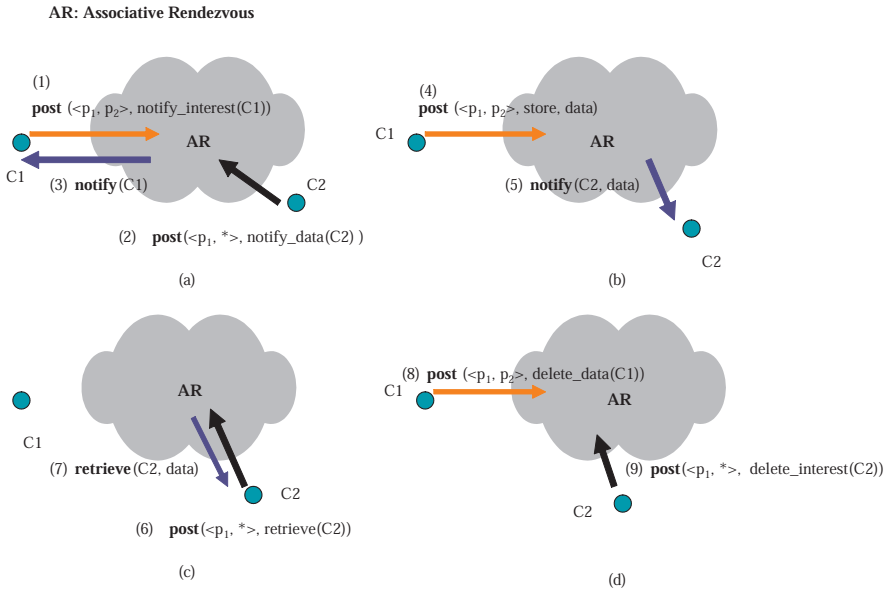

Figure 3.4: An example illustrating Meteor's associative rendezvous and reactive behaviors

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 3.4. 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.

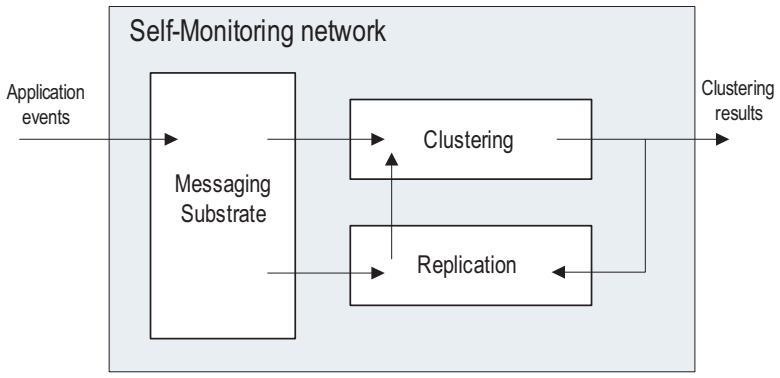

Figure 3.5: Architectural overview, showing how application events are distributed to the clustering module and replicated to ensure robustness to failures. The replication mechanism uses the results from the clustering algorithm to selectively replicate the incoming application events.

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

3.2 Robustness Mechanisms

Figure 3.5 shows an overview of the architecture of the proposed monitoring and data analysis framework. In the figure, the messaging substrate consists of the stack described in the previous section. The clustering module performs the required clustering analysis and will be described in detail in Chapter 4. Both functionality from the messaging substrate and results from the clustering module are used to ensure robustness to failures at different levels. The selective replication mechanism, described in Section 3.2.3, will be evaluated in the next chapter in Section 4.2.3.

3.2.1 Overlay Level

The overlay middleware must guarantee routing of messages and maintenance of the overlay structure despite membership changes and node failures. Most common structured overlays ([81, 71, 74]), provide such self-healing mechanisms, and these can be leveraged for ensuring the robustness of the clustering application. For example, Chord can route around failures by using alternative entries from its routing tables. Furthermore, Chord nodes perform what is called a stabilization mechanism by periodically checking the links in their tables and updating links that are outdated due to structural changes (node joins and departures) and failures. Because clustering is also a periodic task, we must ensure that the frequency of the overlay's self-healing tasks is enough to detect and recover from failures within a single clustering period.

3.2.2 Platform Level

In order to support the replication of data points, as well as the recovery of these points in the event of the loss of the original data due to failure, the overlay platform needs to provide three basic mechanisms. The first is, of course, a primitive that receives a point to be replicated and stores that point at a remote node. The second is a mechanism to recognize the absence of a node and to notify this to the remote site(s) at which its points have been replicated. Finally, the platform needs to provide a way to deliver the replicated points to the application when failures occur.

We take advantage of the dynamic hash functions used in DHTs to realize these mechanisms. This makes the selection of remote replication nodes straightforward because the selection can be made simply by using the dynamic DHT mapping after removing the failed node. We thus define in general terms a node's successors(s) in the overlay as the node(s) to which the keys that correspond to the data at the original node would be mapped in the absence of the original node[1]. With this selection criteria, the realization of the second and third mechanisms also becomes straightforward, given that most overlays readily provide mechanisms to detect the failure of neighbors and that,

[1]This definition is more general than that of a successor in the Chord specification [81], which is in fact a successor in the present sense.

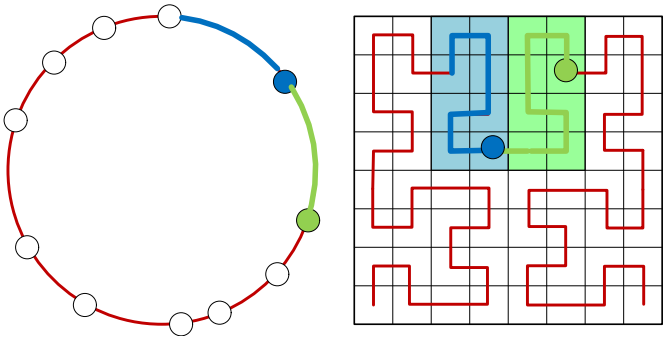

Figure 3.6: In the figure, the points that correspond to the blue region are stored at the blue node and replicated at the green node (and possibly subsequent nodes). If the blue node fails, the blue region becomes the responsibility of the green node, which will add the replicated points to its own and assume the entire region.

once a node fails, the node at which its data is replicated can automatically process this data, without the need for additional messaging overhead.

Once again, we use the Chord overlay as an example to illustrate these mechanisms. As Figure 3.6 shows, in Chord a node is responsible for keys that map to the index space between its predecessor and itself. Therefore, a Chord node has a single successor in the ring topology. When a node fails, its successor learns of this failure by the same stabilization mechanism (described in Section 3.2.1) used to update its routing table. Subsequently, when a successor learns of its predecessor's failure, all it must do is deliver the predecessor's points to the application level, because they now correspond to its own region (see Figure 3.6).

The tradeoff for the simplicity and low cost of this replication scheme is that the proximity of the nodes in the overlay makes the scheme vulnerable to localized failures of multiple nodes. In the next section we show how this tradeoff can be reduced by replicating at multiple nodes rather than just at the single successor, while minimizing the overhead of full replication by using failure probabilities and taking data relevance into account.

3.2.3 Analysis Level (Selective replication)

In order to bound the probability that all replicas of a point are lost given the possibility of multiple node failures, it is necessary to provide the capability of replicating points

at more than one node. Given failure probabilities or rates for system nodes, it is necessary to calculate a value r corresponding to the number of replicating nodes for each point, taking into account the probability that at least one node will not fail. Thus, if a point is replicated at r nodes, we can be fairly certain that the point will be received by an instance of the application (in this case, clustering) running at one of these nodes. Because we would like to preserve the simplicity and low overhead of the replication scheme as described in the previous section, replicating a point at r nodes can be accomplished by replicating the point at successive successors. In Chord, this would correspond to a chain of r successors starting at the original node. We refer to this as a replication chain of length r.

A straightforward replication model, which we call full replication, is to always replicate a data point at a pre-determined (fixed) number of nodes r^*. This number could be calculated by assuming a maximum or average failure rate across all system nodes and obtaining the minimum number of replication nodes that satisfy a desired loss probability. This replication scheme, where all replication chains are of length r^*, can have excessive overhead, especially if it is necessary to consider adversarial fault models in which failure probabilities depend on the data stored at each node. These fault models make it necessary to factor a larger failure rate than average in the calculation of r^*, and thus many points will be replicated at more nodes than necessary under normal system operation.

The particular nature of the clustering application is such that loss is tolerable because events are periodically being generated and the preservation of individual event values is not necessarily as important as preserving patterns of interest, as well as outlying values. Unlike typical data storage systems, not all data items are equally important: if the main goal is to detect outliers from the clustered data, it is more important not to miss a single outlier value due to loss than to miss a few of the clustered points. For the same reason, nodes that store important points may have higher failure rates due to adversarial attacks. Thus, some points will need to have more replicas than others, while some might not need to be replicated at all.

The problem is then to choose a chain length $r_i \geq 0$ for each data point i, for

which we propose two probabilistic replication models. In both models, replication decisions are based on two main factors: 1) the probability that the node j at which the point is stored will fail; 2) the likelihood that the given point i will be recognized as an outlier by the clustering algorithm. These quantities are denoted by p_j and a_i respectively. We will first describe the basic mechanism of both models, and then derive and compare their loss probabilites and replication chain lengths, using full replication as a benchmark.

Replication models

The first replication model, referred to as first node replication (FNR), relies on directly calculating a point loss probability when a node in the overlay receives a point, which is given by the probability that the number of consecutive node failures starting at the local node is greater than the number of replicas of the given point. Let F be the random variable that corresponds to the number of consecutive node failures. The chain length r_i is calculated directly at the first node that receives the point as the smallest value that satisfies:

$$P[F > r_i] < \alpha, \tag{3.1}$$

where α is an upper bound on the loss probability. Once the value of r_i is obtained at the first node, the point will be replicated at this number of successors without further calculation. The outlier likelihood a_i can be factored into this model in two ways. If it is expressed as a probability (a value between 0 and 1), it can be combined with p_j to obtain a new failure probability for the node that depends on both. Alternatively, it can be used to determine different values of α (lower for more important points). This model has the advantage that it is simple to implement and has inherent guarantees on data loss, but it ignores failure probabilities at successors of the first node that could affect the loss probability. Thus, it is suitable for environments where nodes have relatively uniform failure rates.

In the second model, called multi-node replication (MNR), each node j in the replication chain decides whether or not to replicate the point further based on a replication

probability q_j calculated as a function of a_i and the values of p_j along the replication chain. Node j will continue to replicate the point if the outcome of a binomial test with probability q_j is positive. Depending on the way that q_j is derived, different bounds on loss probability can be obtained, so that the implementation of this model is more heuristic in nature than FNR. However, it is meant to be more reliable in cases with variable failure probabilities across nodes, as with adversarial attack models that target nodes based on the data stored in them.

In order to compare both models, we will derive a general expression for the loss probability. Given a bound on this probability, we will also compare expected values for the replication chain lengths and compare them to a full replication approach. These theoretic results will then be verified empirically in Section 4.2.3. In both cases, we model the number of consecutive failures F as a geometric random variable with probability p (the probability that a given node in the chain will fail). This is:

$$P[F = f] = p^f \cdot (1 - p) \tag{3.2}$$

If, in general, r is the number of replicas of a given point (not counting the original), then the probability of loss in FNR is:

$$P[F > r] = \sum_{f=r+1}^{\infty} p^f \cdot (1 - p) = 1 - \sum_{f=0}^{r} p^f \cdot (1 - p) = p^{r+1} \tag{3.3}$$

Because FNR uses this failure probability directly in equation 3.1, equation 3.3 gives a way to calculate the chain length r:

$$\begin{aligned} P[F > r] &< \alpha \\ p^{r+1} &< \alpha \\ (r+1)\log p &< \log \alpha \\ r &> \log_p \alpha \end{aligned} \tag{3.4}$$

The change of the inequality in equation 3.4 is because p is a probability, and thus between 0 and 1, so that the log is negative.

Obtaining an expression for the loss probability for MNR is more complex because the number of replications is also a random variable R. For this analysis, we make two simplifying assumptions. First, the distribution of the number of consecutive failures is assumed to be as in equation 3.2, with a uniform failure probability across nodes. Likewise, the replication probability is assumed to be of the form $q = n \cdot p$, also uniform across nodes, so that R is also distributed as a geometric random variable. Even though the MNR model is meant for variable failure probabilities, this analysis is still illustrative as a simple application of the model that can still account for an adversarial attack that affects p directly from a_i. Additionally, it will allow observing loss probability bounds and replication chain lengths for different choices of n as a basis for defining a heuristic for MNR.

To calculate the loss probability $P[F > R]$, it is necessary to obtain the joint probability distribution of F and R. Since these are independent events, this is:

$$P[F = f; R = r] = p^f \cdot q^r \cdot (1-p)(1-q) \tag{3.5}$$

Now, the loss probability is given by:

$$\begin{aligned}
P[F > R] &= \sum_{f=1}^{\infty} \sum_{r=0}^{f} p^f q^r (1-p)(1-q) \\
&= \sum_{f=1}^{\infty} p^f (1-p) \sum_{r=0}^{f} q^r (1-q) \\
&= \sum_{f=1}^{\infty} p^f (1-p)(1-q^{f+1}) \\
&= (1-p) \left[\sum_{f=1}^{\infty} p^f - q \sum_{f=1}^{\infty} (pq)^f \right] \\
&= \frac{p - p^2 q - pq^2 + p^2 q^2}{1 - pq} \qquad (3.6) \\
&= \frac{p - np^3 - n^2 p^3 + n^2 p^4}{1 - np^2} \qquad (3.7)
\end{aligned}$$

Equation 3.6 follows from the fact that pq is less than 1, because both p and q are probabilities. Equation 3.7 is obtained simply by replacing q by np. Table 3.1

	p					
n	0.01	0.02	0.03	0.04	0.05	0.06
1	0.00999	0.01999	0.02997	0.03993	0.04988	0.05979
2	0.00999	0.01996	0.02989	0.03975	0.04952	0.05918
3	0.00999	0.01992	0.02976	0.03944	0.04892	0.05815
4	0.00998	0.01987	0.02957	0.03901	0.04808	0.05670
5	0.00997	0.01980	0.02934	0.03845	0.04699	0.05483
6	0.00996	0.01971	0.02905	0.03776	0.04565	0.05252
7	0.00995	0.01961	0.02870	0.03695	0.04407	0.04979
8	0.00993	0.01949	0.02831	0.03601	0.04224	0.04662
9	0.00991	0.01936	0.02786	0.03495	0.04015	0.04300
10	0.00990	0.01921	0.02735	0.03375	0.03782	0.03893

Table 3.1: Loss probabilities for MNR replication model obtained from applying different values of p and n to equation 3.7

shows different values of loss probabilities obtained using equation 3.7 for different values of p and n. Notice that until a failure probability of 6%, which is quite high, the loss probability stays under 0.05, which is within the error bound of the clustering algorithm. A simple way to choose the replication probability q_j at each node is to make it equal to p_j, which corresponds to setting $n = 1$. As can be seen from the table, this is suitable for p up until 5%. Another way to choose the replication probability is to use the values in the table to choose the smallest n that meets a required bound on the loss probability. This is equivalent to setting q_j to a higher value after a_i (outlier likelihood) is combined with p_j, so that it is always possible to assume $n = 1$. This way, a low error bound is maintained for local failure probabilities of up to 50% with an adversarial attack model. This is the heuristic used in Section 4.2.3 to evaluate the MNR model and verify its usefulness.

Both probabilistic replication models are meant to achieve a high degree of reliability while maintaining the overhead of replication low when compared to a full replication model. This means that the length of the replication chains must be short on average. Figure 3.7 shows replication chain length values for the three replication models that guarantee a loss probability bound of 5% under different probabilities of failure. For full replication, we assume a maximum failure probability of 50% as a worst case to guarantee the loss probability bound in all cases. For FNR, the chain lengths are obtained directly from equation 3.4, and for MNR they are obtained as an expected

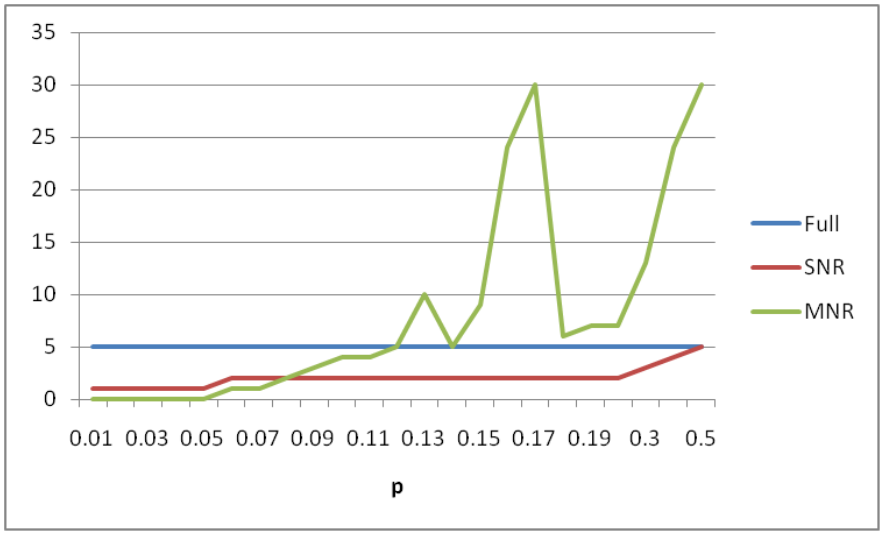

Figure 3.7: Replication chain lengths estimated for different replication models and probabilities of failure.

value, setting q to the smallest multiple of p that achieves the desired bound.

It is interesting to observe that up until 8% failure probability, the expected chain length of MNR is close to zero and shorter than both FNR and full replication. This can lead to significant overhead reduction when failure rates are low and a small percentage of the points are expected to be outliers and worth preserving. However, after $p = 0.12$, the expected chain length surpasses that of full replication and its value becomes erratic. FNR provides a good compromise for cases where the failure rate is high, and, as stated earlier, relatively uniform. Recall, however, that the loss probability model used for MNR is a simplification, and heuristics can be employed to obtain the benefits and good response to variable and/or adversarial failure probabilities without exceeding a maximum replication chain length. One technique, used for the evaluation in Section 4.2.3, is to reduce the value of q_j obtained using the MNR model proportionally to the current length of the replication chain.

Calculating failure and outlier probabilities

During the lifetime of the system, the current probability of failure for a given node is updated using a Bayesian conditional probability, where the probability of failure is conditioned by the time elapsed since the last failure of the node. This is:

$$P\left[Fail|T \geq t\right] = \frac{P\left[T \geq t|Fail\right] \cdot P\left[Fail\right]}{P\left[T \geq t|Fail\right] \cdot P\left[Fail\right] + P\left[T \geq t|\overline{Fail}\right] \cdot P\left[\overline{Fail}\right]}$$

Here, $P[Fail]$ is a statically defined probability of failure that could be parameterized by load and/or other factors according to the failure model. To condition the distribution of the time that a node operates without failure by previous failures of the node, we use two average times that can be determined from the failure model and adjusted with the operational history of the system. The first is the mean time to failure (MTTF), which is a parameter for the probability when a node has not yet failed ($P\left[T \geq t|\overline{Fail}\right]$), that only considers random failures or failures due to normal operational malfunction. Once a node has failed, its operating time is distributed according to the mean time between failures (MTBF) of nodes that could be heavily loaded or targeted by malicious users due to their importance or weakness.

A Bayesian conditional probability is also used to calculate the probability that a particular point i is an outlier. In this case, the event of an outlier is conditioned on the distance of the event to the closest of the centroids of clusters found in a previous iteration. The intuition is that if a point is close to the centroid of a previously detected cluster (as determined by the cluster's width), then it is not likely to be anomalous, in contrast to a point that is far from existing clusters. The update rule is given by:

$$P\left[A|U \leq u\right] = \frac{P\left[U \leq u|A\right] \cdot P\left[A\right]}{P\left[U \leq u|A\right] \cdot P\left[A\right] + P\left[U \leq u|\overline{A}\right] \cdot P\left[\overline{A}\right]}$$

Here, $P[A]$ is simply the frequency of outliers found at node j in previous iterations with respect to the total number of points received at the node. This way, both previous outliers and clusters are used to determine outlier probabilities, the former accounting for outliers that have been seen before and the latter for those that may not have been observed, but that deviate from normal behavior. In the formula above, instead of using the absolute distance of the point to existing clusters, the probability is conditioned on a ratio U of the distance to the corresponding cluster width. This is to account for clusters of different densities and widths. Given cluster centroids c_k and their corresponding widths w_k, the actual ratio u calculated for each point p is given by:

$$u = \min_k \left(\frac{dist(p, c_k)}{w_k} \right)$$

The conditional probability distributions of the distance ratios can be obtained in two ways. One is simply to define static distributions that favor ratios smaller than one for cluster points ($P[U|\overline{A}]$) and larger than one for outliers ($P[U|A]$). The other is to maintain frequency distributions by calculating u for outliers detected by the algorithm and for cluster points.

Chapter 4

Decentralized Online Clustering

This work is based on the premise of realizing online and decentralized data analysis, exploiting the collective computing resources of distributed systems. Online analysis because offline analysis tends to produce models that do not capture short-term or transient system behavior. Decentralized because, although highly accurate, centralized approaches are infeasible in general because of the costs of centralization in terms of infrastructure, fault-tolerance, and responsiveness. However, it is important to show that the tradeoff that must be made for decentralization in terms of accuracy and network resources is worth the aforementioned advantages.

This chapter describes the decentralized online clustering (DOC) algorithm in detail, its implementation based on the robust content-based messaging infrastructure presented in Chapter 3, specific application details, and a comparative evaluation with well-known density-based clustering algorithms. The experiments will show that DOC is close to these centralized algorithms in terms of accuracy, in addition to being less sensitive to changes in its input parameters, which is important for online, automated application. They will also show that, because of decentralization, the running time of DOC scales well with the size of the data set to be analyzed, when considering the impact of centralizing data elements that are originally distributed.

4.1 Algorithm Description

As mentioned in the introduction, this research targets networked systems in which individual components can monitor their operational status or actions, represent them using sets of globally known attributes, and periodically publish this status or interactions as semantic events that contain a sequence of attribute-value pairs.

These events correspond to the data elements to be clustered, each one of which is represented as a *point* in a multidimensional space, as shown in Figure 4.1. Each

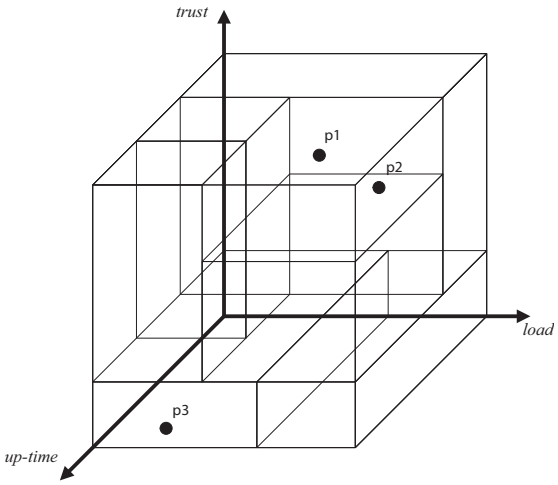

Figure 4.1: Example of an information space with three attributes. Point $p3$ can be considered an outlier with respect to the relatively denser cluster formed by points $p1$ and $p2$. The space is divided into regions, within which clustering analysis is performed individually.

dimension in this space, referred to as an *information space*, corresponds to one of the event attributes, and the location of a point within the space is determined by the values for each of its attributes. It is assumed that the range of values of each attribute is an ordered set. For each set, a distance function can be defined in order to measure the similarity between points (i.e. similarity is inversely proportional to distance). This definition is straightforward for quantitative attributes, and can be applied to non-quantitative attributes as well with an appropriate encoding.

The notion of similarity based on distance in each dimension extends to the multidimensional information space, for which a distance function can also be defined in terms of the unidimensional distances. Conceptually, a cluster is a set of points for which mutual distances are relatively smaller than the distances to other points in the space [35]. However, the approach for cluster detection described in this paper is not based primarily on evaluating distances between points, but rather on evaluating the relative *density* of points within the information space. In this case, point similarity is directly proportional to point density.

The approach used for the evaluation of point density, and thus for the detection of clusters and outliers, is sketched in Figure 4.2. The idea is to divide the information

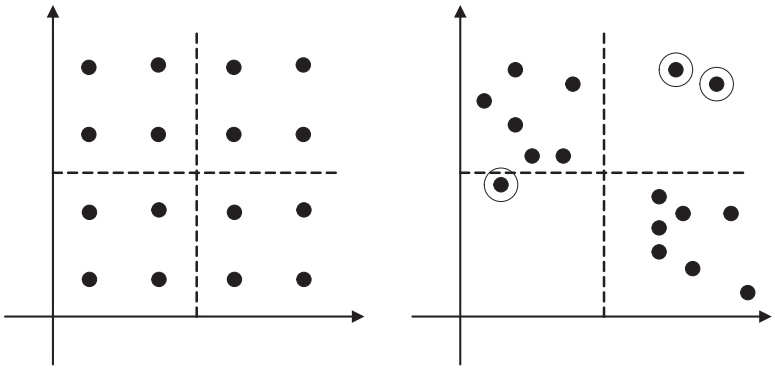

Figure 4.2: (a) Uniform distribution of data points in the information space. (b) Point clustering and outliers among regions; the circled points are potential outliers.

space into *regions* and to observe the number of points within each region. If the total number of points in the information space is known, then a baseline density for a uniform distribution of points can be calculated and used to estimate an expected number of points per region (Figure 4.2 (a)). Clusters are recognized within a region if the region has a relatively larger point count than this expected value, as in Figure 4.2 (b). Conversely, if the point count is smaller than expected, then these points are potential outliers. However, clusters may cross region boundaries, as shown in the lower left quadrant of Figure 4.2 (b), and this must be taken into account when verifying potential outliers.

The approach described above lends itself to a decentralized implementation because each region can be assigned to a particular *processing node*. Nodes can then analyze the points within their region and communicate with nodes responsible for adjoining regions in order to deal with boundary conditions. We assume that the information space is predefined and globally known; however, the assignment of regions to nodes need not be static, and can thus be adapted to the arrival and departure of processing nodes in the network, as long as a mechanism exists to map data points to the node responsible for the region in which the point is located (we describe an implementation of this mechanism in Section 4.1.3). Note the distinction between processing nodes, physical nodes, and data points in the space. Physical nodes host processing nodes, so that multiple processing nodes may correspond to a single physical node. However, we they are often the same and we refer to them indistinctly, unless explicitly referring to

a physical node. Data points correspond to status or interaction events of distributed system components. The status of a distributed component may be mapped to any node, depending on the region of the space where the point lies. In fact, the physical sets of machines corresponding to nodes and components may be completely or partially distinct. Henceforth we use node to refer to a processing node and point to refer to an status or interaction event (set of attribute values). The following section presents a formal description of the decentralized clustering algorithm that is based on the concepts described in this section.

4.1.1 Specification

The decentralized online clustering (DOC) algorithm is based on a finite, discretized and normalized information space $S \subset \mathbb{Z}^{+^a}$, where a is the number of dimensions (attributes). The space is discrete to facilitate the application of the mapping described in Section 3, and it is normalized to enable the definition of a single unit of distance within the space. To convert an arbitrary space into the required form, each application must define, for each dimension i, a distance d_i that will correspond to a unit distance in the normalized space. The unit distance is a general property of each attribute, as determined by the application, and is the minimum difference in attribute values that should be distinguishable (i.e. a difference of less than d_i original units is considered negligible). The range of each dimension will thus be the interval $[0, x_i]$, where $x_i = (max_i - min_i)/d_i$.

The following definitions correspond to the concepts introduced in the previous section and to the main algorithm elements and operations:

- *Contiguous points*: Two points are contiguous in S iif if they differ by at most one unit in at most one dimension.
- *Region*: A region is a set $R \subseteq S$, such that any two points in R can be connected by a sequence of contiguous points in R. Each node is responsible for a unique region, denoted by node.region.
- *Neighbor*: Two regions R_1 and R_2 are neighbors iif $\exists p_1 \in R_1$ and $p_2 \in R_2$ st. p_1

```
COLLECT_DATA (T)
1  listen for period T
2  points <- {p | p received during T}
END
```

Figure 4.3: Algorithm for the data collection phase at each node

and p_2 are contiguous points. Two nodes n_1 and n_2 are neighbors if n_1.region is neighbor of n_2.region.

- *Spatial density* (density(R, X)): The spatial density of a region R with respect to a point set $X \subseteq R$ is the ratio size(X)/size(R), where size() denotes the cardinality of a set. If the region is omitted for a set Y, density(Y) is the spatial density with respect to Y of the smallest region R_Y such that $Y \subseteq R_Y$.

- *Linear density* (density(d, X)): The linear density of a point set X along a dimension d is the ratio of size(X) and the maximum distance between points of X along dimension d.

- *Cluster*: Let D be the data set being analyzed, then a set of points C is a cluster with respect to a region R, if density$(R, C) >$ density(S, D).

- *Centroid of a set* (centroid(X)): Point with minimum average distance to each of the points in X.

- *Width of a set along a dimension* (width(X, d)): The maximum distance of any point in X to centroid(X) along dimension d.

- *Bounding box of a set* (box(X)): Minimal region that contains all of the points in X, determined by the width(X, d) for each dimension d.

The algorithm is divided into two phases, 1) the data collection phase and 2) the analysis and consolidation phase. Figure 4.3 shows the steps of data collection. Basically, nodes are synchronized by a time period T, during which data points are exchanged over the network and received by processing nodes.

Figure 4.4 shows the algorithm for the analysis and consolidation phase. It shows how a node performs the clustering analysis for the points received during the period

```
ANALYSIS (R, size(D))
1  exp_count = size(R) * (size(D) / size(S))
2  clusters <- {}
3  outliers <- {}
6  count = size(points)
7  If count > exp_count
8    Add (centroid(points), box(points)) to clusters
9  Else
10   outliers <- points
11 For (c, b) in Changed(clusters)
12   Broadcast((c, b), extended(b))
13   remote <- ReceiveBroadcasts()
14   clusters <- Merge(clusters, remote)
15   outliers <- RemoveContained(clusters)
16 Loop
END
```

Figure 4.4: Simplified algorithm for the analysis/consolidation phase at each node

of the preceding data collection phase. The analysis consists of comparing the number of points received with an expected number of points (line 7). Because the expected number of points is calculated (line 1) based on the size of the region relative to the point density of the information space, given a particular data set size, this comparison actually applies the definition of a cluster given above.

If the point count is higher than the expected number of points, then these points form a cluster in the region under consideration. When it finds a cluster, the node records its centroid and bounding box (line 8). Conversely, when the point count at a particular node is not as large as expected, the points received in the corresponding region are potential outliers. Before they are labeled as such, however, the boundary conditions must be checked in case there are nearby clusters at neighbors to which the potential outliers might actually correspond.

Once processing nodes have determined the existence of clusters or outliers locally, it is necessary to consolidate the data from single clusters that may span multiple nodes in order to obtain a single set of descriptors for each cluster. There are three possible cases for each cluster in the data:

A. The cluster is entirely contained within a processing node's region.

B. The cluster is, for the most part, in a single region, but the remaining points

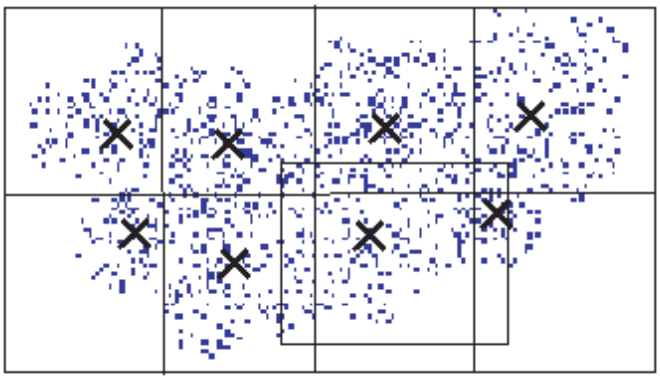

Figure 4.5: Initial clustering result and cluster broadcast at node 7

that lie in neighboring regions are treated as outliers. This is the case shown in Figure 4.2.

C. The cluster lies across several nodes' regions, so that each node identifies the points within its region as clusters and calculates centroids separately, as shown in Figure 4.5.

Also shown in Figure 4.5 is the grid of processing node regions (referred to by node number 1 through 8). Each node that contains a local cluster broadcasts its description to neighboring nodes (line 12), using a range determined by the size and density of its own cluster, as follows. Each cluster C has a bounding box b, the size of which is extended by adding to width(C, i) for each dimension i. First, δ_i = density(i, C) is calculated. The inverse of this density is the average distance between points along that dimension. Thus, for a given constant k, $k \times \delta_i^{-1}$ is the extension of the box along dimension i. This means that, on average, k more points of the same cluster should be contained in the extended box, assuming that the cluster has more points than those contained in the original region. Thus, increasing k, the likelihood that the extension of the cluster region along that dimension crosses the boundary is increased. We have determined empirically that a value of $k = 3$ produces good results. Figure 4.5 shows the query issued by node 7.

In case A above, it may be that the broadcast will not reach other nodes because the extended bounding box did not intersect their regions. The nodes whose regions

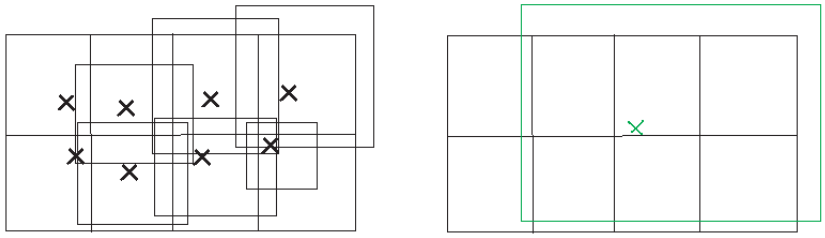

Figure 4.6: Node 7 after one level of agglomeration

are intersected (nodes 2, 3, 4, 6, and 8 in the figure) will receive the broadcast and merge the cluster information received with their own findings (lines 13, 14). Points that correspond to case B will be removed from the list of outliers to be reported in the node's result and added to the new cluster (line 15). In case C, the two parts of the cluster are merged and a new centroid and extended bounding box are calculated. Figure 4.6 shows how, at node 7, all of the broadcasts received from neighboring nodes are merged into a single cluster. Notice that the new centroid may not necessarily fall in the region of the node where the new cluster is obtained. Only the node that "owns" the final resulting centroid (in whose region the centroid falls) will report the corresponding consolidated cluster. Nodes will keep broadcasting cluster data for any cluster that is extended in this way, until no further broadcasts are possible (no clusters have changed, as indicated in line 11).

4.1.2 Algorithm Application

This section discusses the considerations and conditions required for the application of the decentralized clustering algorithm. In order for this approach to work effectively, first of all we require a mechanism to map points to a dynamic set of nodes, so that each node receives the points located in its corresponding region. Our implementation of the decentralized clustering algorithm builds on the messaging infrastructure described in Section 3, which efficiently provides the required mapping as well as a platform for monitoring, communication, and coordination between system and processing nodes. Other application considerations follow.

Parameter Estimation

Autonomic monitoring implies that the clustering algorithm has to be applied with minimal intervention and a-priori knowledge about the data being clustered. This ensures both that the algorithm can be deployed easily for new management tasks or information spaces and that it can adapt to significant changes in the data. As will be demonstrated in the sensitivity evaluation in Section 4.2.1, one of the most important aspects that determine the accuracy of the clustering algorithms is setting the input parameters for the analysis.

Because clustering is an underdetermined problem, there is no general best configuration for the parameters of any clustering algorithm. Instead, depending on the algorithm, effectively setting the parameters will depend to a greater or lesser extent on the characteristics of the data, the information space, and the application. The more dependent the parameters are on the data, however, the more need for reconfiguration of the parameters when the data changes. Because the main design goal of DOC is to make it suitable for autonomic application, we propose an automated method for parameter estimation that requires minimal initial input and has minimal dependency on the particular characteristics of the data when these are unknown.

Density-based clustering algorithms such as DBSCAN and Clique use a density threshold (a number of points per unit-space) to determine if points in particular regions of the space are clusters or not. This density threshold is therefore a necessary parameter for all such density-based algorithms. However, each algorithm has its own way of specifying the threshold, usually decomposing it into a form of its two constituent factors. DBSCAN, for example, requires the specification of a minimum number of points ($MinPts$) that can be considered as a cluster, along with a minimum distance or radius (EPS) from a point where that number of points can be found (see Section 2.2.3 for a more detailed description). Notice that the latter parameter, the distance, is a data-dependent parameter, since clusters can exist with the minimum number of points, but with a greater distance between points than the one specified.[1] In fact, DBSCAN

[1]We are assuming that the information space is defined so that the minimum unit is significant to the application, as explained at the beginning of Section 4.1.1; otherwise, significance to the application

provides a manual method for analyzing the data before the algorithm is applied in order to obtain this parameter.

The way that each of the parameters is calculated for the algorithm, along with the algorithm design, induce a cluster definition — i.e. the types of the clusters that the algorithm will, and will not, detect. The definition induced by density-based algorithms like DBSCAN and Clique assumes that the region size necessary to cover the cluster of minimum density in the data is known or can be estimated. We argue that without a-priori knowledge of the data distribution, a less stringent cluster definition is required. DOC defines a cluster as a set of at least *MinPts* points in a region of the space whose density is greater than that of a uniform distribution of the points in the space (note that this corresponds to the formal definition presented in Section 4.1.1). This definition translates to a parameter specification that includes the minimum number of points per cluster (as in DBSCAN) and the total number of points N being clustered. To see the latter, note that:

$$uniform\_density = \frac{N}{space\_size}$$

To define the above parameters for DOC, the user requires minimal data-dependent information, namely the number of points $N = size(D)$, which is either given or can be obtained from the number of nodes (which, if unknown, can be obtained even in the face of high churn [5]), event generation rate, and collection period T. The minimum number of points per cluster is purely application specific and should not change often during the lifetime of the application. Or, if it depends on N, it can be expressed as a percentage (as is done in Clique [4]). The algorithm must still estimate a region size to divide the space among the processing nodes, however. Although our relaxed definition allows any division, by calculating the expected number of points in the region from the uniform density, our previous evaluations [69] showed that the region size has a significant impact on accuracy (which is precisely why it is required by existing algorithms). We therefore obtain the region size as that of a region R such that:

---

would help determine the minimum distance as an application-dependent parameter.

$$size(R) \times uniform\_density = MinPts \tag{4.1}$$

This is the region size for which the expected number of points in a uniform distribution equals the minimum number of cluster points specified by the parameter $MinPts$. To justify this choice, we make two observations. First, a larger region size risks grouping points that according to the definition should be marked as outliers. Second, a smaller region size risks missing cluster points that are grouped with a density that is close to the uniform density. Therefore this is the smallest region size that conforms to the given cluster definition. Furthermore, for dense information spaces (informally defined as those in which $N >> MinPts$ and $N > size(R)$) this region size is likely to be smaller than the sizes of the clusters to be found, which we found in [69] to be important for accuracy.

Of course, as stated earlier, with no assumptions on data distribution, it is not possible to prove anything about the optimality of the region size estimation. However, the proposed parameter estimation can easily be applied in an automated fashion (even to other algorithms like DBSCAN and Clique), and, as Section 4.2 will show, yields results that are comparable to those obtained with more informed parameter estimations. Furthermore, the same automated mechanism can be applied with knowledge of the data distribution to obtain smaller and more refined region sizes where data is known to cluster (because a smaller region size is expected to contain $MinPts$), and, conversely, larger region sizes in regions where no clusters are expected.

Online Execution

Applying the decentralized clustering algorithm online implies that it is possible to collect data and produce results that can be used for management in a timely manner. In a system where monitoring is done continuously, clustering analysis can be alternated with data collection, as shown in Figure 4.7 (a). To be able to effectively apply the algorithm in this way, the time required for the analysis phase must be short in comparison to the data collection phase, so as to minimize the amount of data that is not collected while the analysis is being performed. It is also possible to run the analysis phase in

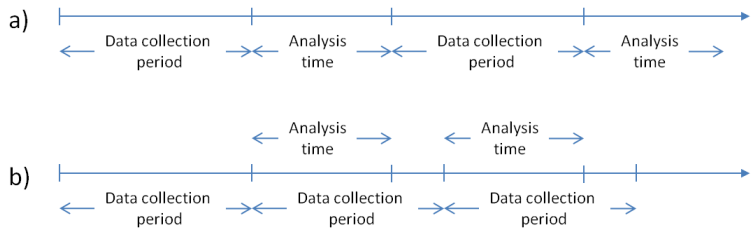

Figure 4.7: Clustering online execution timelines with (a) alternating and (b) overlapping data collection and analysis phases.

parallel with data collection, as shown in Figure 4.7 (b). As long as the analysis time does not exceed the collection period, clustering can proceed in this way, always with the data collected in the previous period.

The preceding cases appear to assume that the results for every different collection period are produced independently of previous data. However, it should also be possible to cluster data incrementally, after an initial run of the algorithm, by adding new points to modify previous results (instead of re-clustering from scratch with the new set of points). DOC can be readily applied in this way, with the following three cases for each new point received:

- If a node's region already contains a cluster and the cluster contains the new point, then add the point to the cluster and recalculate the centroid.
- If a node's region contains outliers, but with the new point it satisfies the cluster definition, then calculate the centroid and bounding box and record for consolidation.
- If a node's region is empty, or the new point does not make existing points satisfy the cluster definition, then record as a potential outlier.

Notice that only the second case requires new broadcasts from the receiving node, so that the time for clustering can be significantly shortened. When applying the algorithm in this way, a rolling window can be maintained so that old points are forgotten as new ones are produced, and the total number of points is kept constant.

Load Balancing

As with any distributed application, balancing the load between processing nodes is important for both avoiding bottlenecks and singular points of failure, and maximizing the utilization of all of the available distributed resources. Because of the nature of clustering and the spatial distribution of data that is employed by DOC, however, the problem of load balancing among processing nodes presents special challenges. In other words, by design, points that are clustered in the information space will be sent to and processed by a reduced number of processing nodes.

There are two basic sources of load that must be considered in a DHT-based application like DOC, which we call processing load and query load. The first and most evident is that of processing load, which is the load imposed on a node by the data that maps to it in the DHT, including the processing and storage of this data (if necessary). Balancing processing load is relatively straightforward: it suffices to offload part of this data and/or processing to underloaded nodes, while keeping a local pointer to the new location(s) of the data and/or results so that they are accessible via the DHT interface. Query load, on the other hand, is not as evident or straightforward to manage. It is the load of the messages used to route information, including data and queries, over the overlay topology that supports the DHT. Even if storage and processing are balanced, all of the requests and queries for related data will still have to go through the reduced number of nodes that hold the pointers to this data. Figure 4.8 illustrates this issue.

Applications like DOC are affected to a greater extent by query load than they are by processing load. This is because the processing at each node is very light and the storage requirements even for large numbers of data points are relatively small. In fact, points don't even have to be stored if the counts and cluster information are updated each time a point is received. Query load, on the other hand, is incurred for every point routed over the network and received by a processing node. As mentioned earlier, query load imbalance occurs in DOC because clustered data maps to a reduced number of nodes.

The way to handle query load imbalance is to insert *virtual nodes* in the overlay, in

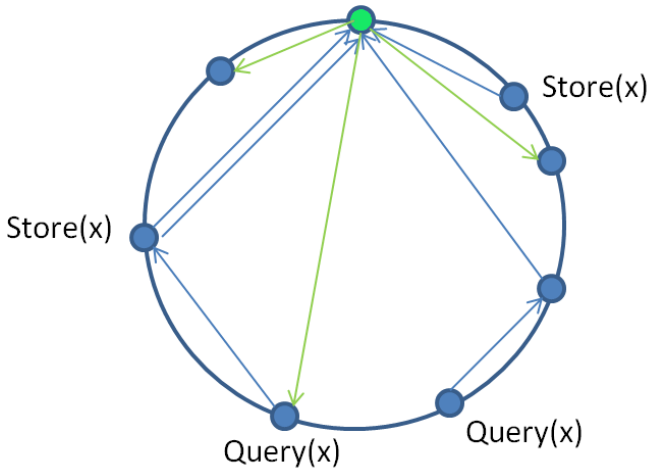

Figure 4.8: Illustration of query load imbalance. Store and query requests for element X are all routed to the highlighted node (blue arrows), even though the data may be stored at other nodes, due to load balancing, as indicated by the green arrows.

the index space of the overloaded node(s). A virtual node joins the overlay just like any other node, so that the routing tables of existing nodes will be updated with the address of the physical machine where the virtual node is hosted. If this machine previously hosted and underloaded node(s), then the virtual node will effectively handle traffic previously mapped to the overloaded node, and thus the corresponding region in the information space.

Creation and deletion of virtual node groups for load balancing is a topic that has been studied in previous work [76]. In DOC, however, there is an additional consideration because of the effect that the division of the information space has on the accuracy of the clustering results [69]. Recall from the previous section that we estimate a region size for each processing node as a parameter of the algorithm. It is likely that this region size does not coincide with the size of the region that maps to each physical host on the overlay. Therefore, each host's region must be split locally into units of the correct size, which Figure 4.9 (a) shows. Each of these local subregions corresponds to a processing node that applies the DOC algorithm. However, for groups of local nodes (co-located at a single physical node), the consolidation is not done with broadcasts, but rather by aggregating results as in the Clique algorithm.

The local subregions are used as units for load balancing to preserve the accuracy

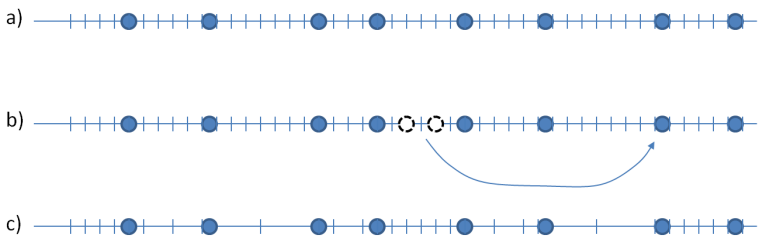

Figure 4.9: Overlay ID space with the region of each node divided into subregions. (a) Initial uniform distribution. (b) Virtual nodes created in an overloaded region. (c) Subregions given known data distribution.

of the algorithm. In order to send a unit to a remote underloaded node, two virtual nodes have to be created, one for either end of the region, as shown in Figure 4.9 (b). Hosting both of these virtual nodes remotely has the additional advantage that it can help further reduce the traffic at the overloaded node because in Chord, most traffic that maps to a node goes through the node's predecessor. In order to reduce complexity arising from hierarchies of virtual nodes (virtual nodes created from virtual nodes), virtual nodes do not create other virtual nodes. Instead, the original node is always responsible for the mapping of its subregions to hosts, and directly receives results from them in order to consolidate them locally. This is also an advantage in the case shown in Figure 4.9 (c), where the region size is recalculated based on a known distribution from previous results. Because it has a reference to all of its virtual nodes, the main node can remove them and redistribute its subregions based on the new region size.

Robustness

Because self-management mechanisms are subject to the same failures that occur on the nodes that they are running on, the robustness of these mechanisms is of great importance to ensure overall system reliability. We are assuming that unbalanced or malicious system utilization is possible, so that we must account for the effect of this utilization on the reliability of the system and ensure the robustness of data analysis, even when node failures or data loss occur because of this behavior. This work gives special attention to ensuring the robustness of the proposed solution at different levels, as part of the infrastructure support for the clustering algorithm. Because of this, a

full section was dedicated to it in the previous chapter (Section 3.2).

4.1.3 DOC Implementation

The DOC algorithm has been implemented in Java as an extension of the implementation of the Meteor content-based messaging and coordination framework presented in Chapter 3. The class that implements the DOC algorithm sits directly on top of the Squid routing engine (Section 3.1.1), which exposes its functionality through the `SquidOverlayService` implementation of the `OverlayService` interface. The associative rendezvous and reactive behavior functionality of Meteor (Section 3.1.2) is reimplemented directly by the clustering module and is wrapped by the `InfoSpace` interface. The functionality of these objects and their interactions will be explained in the rest of this section.

Class Descriptions

An `OverlayService` object is a distributed object that runs on each physical node and implements the following methods:

- `OverlayID join(url)`: Called when the node is initialized. The URL corresponds to the host and port 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 handle these departures to maintain routing capability within the overlay.
- `routeTo(ID, tag, payload)`: Called to send a particular message to one or more nodes responsible for the given IDs. `IDs` is a list of overlay identifiers, `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.

- `subscribe(listener, tag)`: Registers a listener to receive messages sent using the `routeTo` method to the given tag.

- `url[] getNeighborSet()`: Returns the list of addresses that are directly known to the overlay object.

The Squid implementation of the overlay service provides a concrete extension of the `OverlayID` class called `SquidKey`. A Squid key is used to address Squid's multidimensional attribute space, taking either a single value or a value range for each dimension (i.e. a region in the space). As explained in Section 3.1.1, the Squid overlay service uses a distributed routing mechanism to map the region specified in a Squid key to the nodes responsible for the region at that point in time.

The `InfoSpace` interface is an abstraction of the information space from the point of view of the monitoring and clustering services. Points representing monitoring events are inserted into the space, which can then be queried about the existence of clusters and/or outliers. This functionality is provided by the following methods:

- `insertPoint(point)`: Called to insert a point into the information space.

- `getClusters(region, time)`: Returns the descriptors of the clusters, if any, that have a centroid contained in the given region of the information space and that correspond to the given time. Note that cluster points are not returned by this method. Instead only cluster descriptors, which include the cluster bounding box and centroid, are returned.

- `getOutliers(region, time)`: Similar to the above method, but returns any outliers contained in the given region at the given time.

- `getPoints(region, time)`: Returns any points contained in the given region that were inserted at the given time (analysis window). Because a cluster bounding box is implemented as a region in the information space, this method can be used to query the cluster points that correspond to a cluster descriptor returned by the getClusters method.

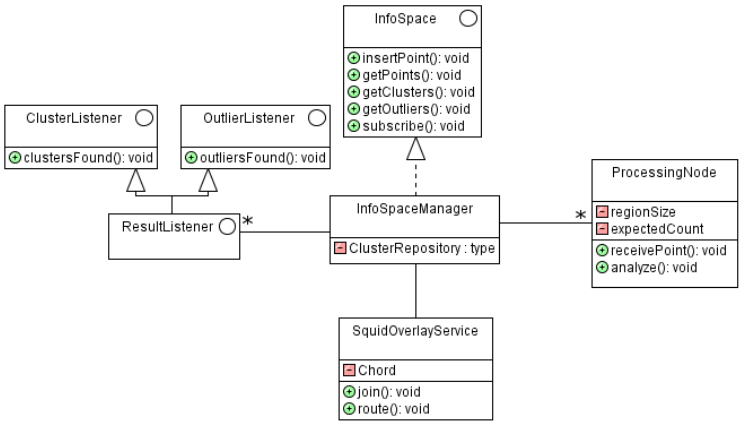

Figure 4.10: Class diagram for DOC implementation

- `subscribe(listener, region)`: Instead of querying the information space for cluster and outlier results, an object can subscribe to the information space as a listener, for results in a particular region or for the entire space. A listener can choose to be notified of clusters, outliers, or both.

The `InfoSpaceManager` is our distributed implementation of the `InfoSpace` interface and which manages the execution of DOC. Figure 4.10 shows a class diagram of the top level of the framework, centered around the `InfoSpaceManager` class. Each `InfoSpaceManager` instance manages the processing nodes that actually implement the logic of the clustering algorithm. Note that there may be multiple processing nodes depending on the number of clustering regions contained within the physical node's region, as was illustrated in Figure 4.9. Users of the `InfoSpaceManager` can subscribe to it as result listeners, to obtain clusters, or outliers, or both, depending on the subscribers' type. The `InfoSpaceManager`, in turn, subscribes as a listener to the squid overlay service and uses it to send point and cluster messages and queries to remote instances of the same class.

Algorithm Initiation and Runtime

When an application creates an instance of `InfoSpaceManager` (henceforth `manager`), this instance in turn creates an instance of the `SquidOverlayService` (henceforth

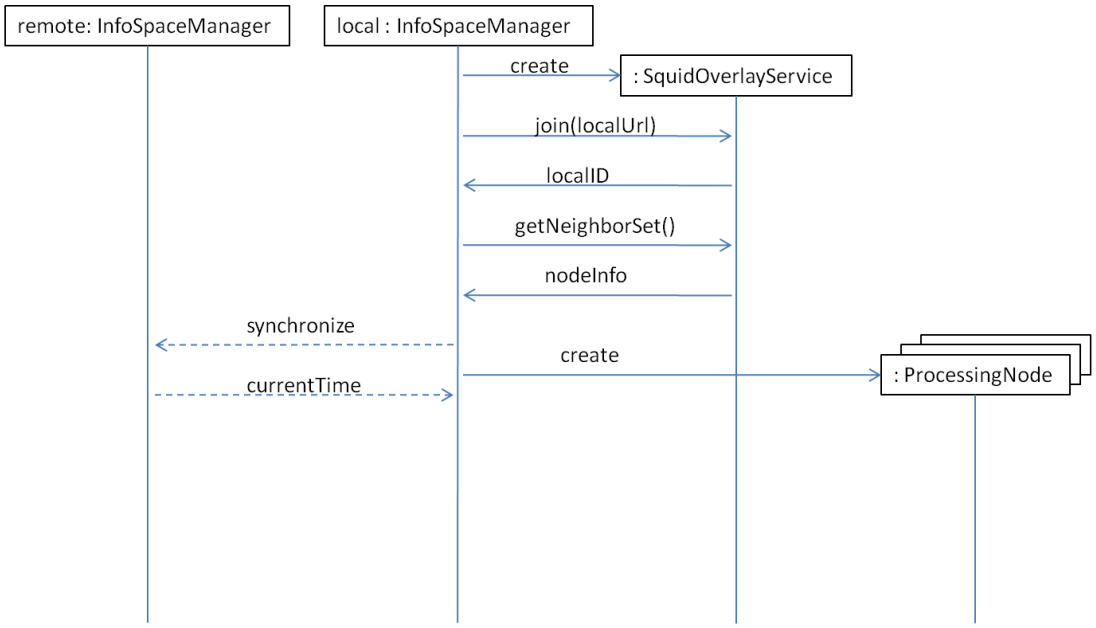

Figure 4.11: Sequence diagram of DOC object creation and initiation. The communication between the local and remote `InfoSpaceManager` instances is done directly using the url obtained through the `getNeighborSet` method, and not using Squid's `routeTo` method.

`squid`), configuring it for routing in an attribute space that corresponds to the multidimensional information space in which clustering will be done. The `manager` object uses `squid` to join the overlay with the physical node's url. Recall from Section 3.1.1 that a new node can join the overlay with the reference of any other node in the overlay, and that Chord's self-configuration mechanisms will automatically take care of updating local and remote routing tables to include the new node. Upon joining, the `manager` object obtains a reference to it's predecessor on the overlay via `squid`'s `getNeighborSet` method in order to synchronize to the current time (analysis window) and calculate the region size, which corresponds to the difference between the local ID and the predecessor's ID (because of the correspondence between Squid's attribute space and the clustering information space). The `manager` can then calculate the clustering region size and create as many processing node objects as clustering regions fit in the physical region size. Figure 4.11 shows this initiation process. Note that the overlay will notify neighbors of the newly joined node, so that they can also adjust the size of their corresponding region and processing node pool if necessary.

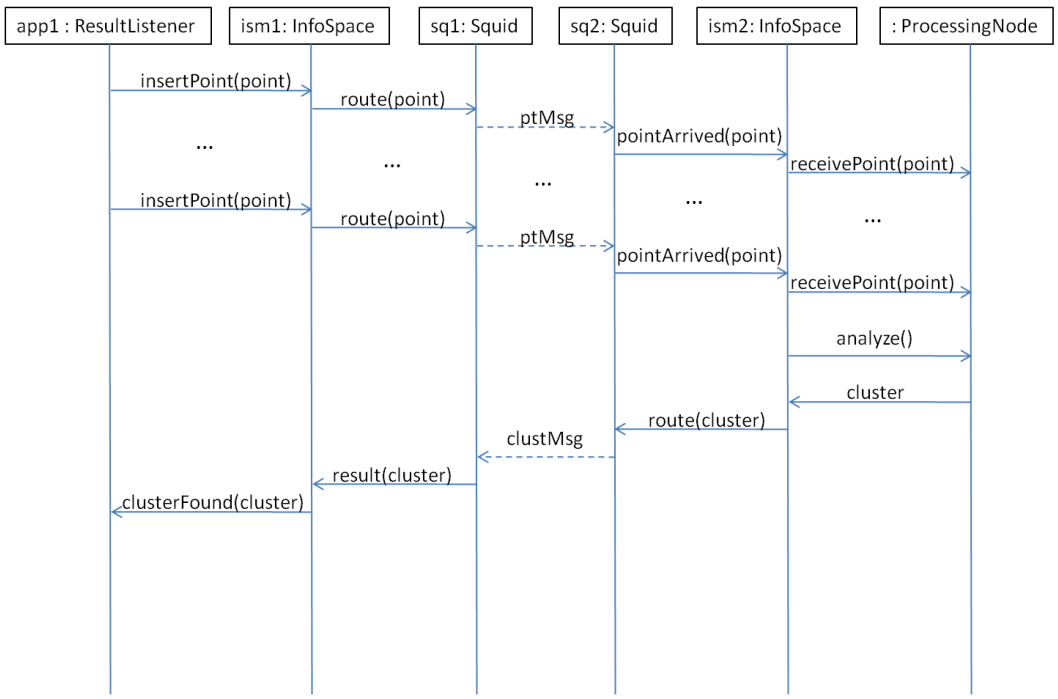

Figure 4.12: Sequence diagram of DOC data collection and analysis. Notice that `app1` acts as both an event source and a result listener.

After a node has been initialized, it can be used to update and query the distributed information space. Figure 4.12 shows a typical sequence for the insertion of multiple points during the data collection phase and the subsequent clustering analysis and result notification. The figure omits the messages exchanged during the cluster consolidation phase, which would take place during the call to `analyze`. Notice that the `routeTo` method of the Squid objects is called both with a points and with clusters (bounding boxes). Actually, the `manager` takes care of creating a `SquidKey` object for each call to `routeTo`, taking into account that a point generates a `SquidKey` without ranges that will be routed to a single remote `manager` instance, while a cluster bounding box generates a `SquidKey` with ranges that will possibly be routed to multiple remote instances.

The same routing mechanism applies when an object subscribes as a `ResultListener` for a particular region: the region is used to create a `SquidKey` that is used to route the subscription to all nodes that manage results for it. In order for the `app1` object in Figure 4.12 to receive the result, it had to subscribe to `ism1` with a region that maps to that corresponding to `ism2`. Figure 4.13 is an illustration of how points and query

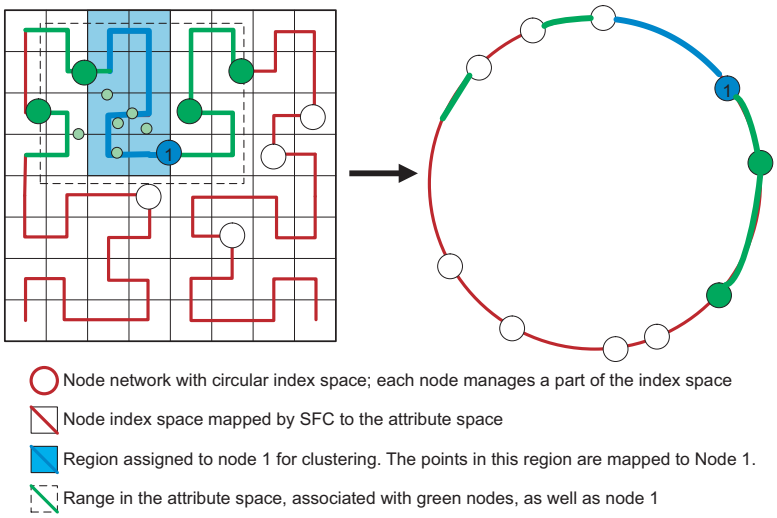

Figure 4.13: Mapping from attribute space to node index space. Matching of cluster points and ranges can be done because intersecting regions are guaranteed to map to a non-empty set of common nodes, such as node 1 in the figure.

regions (subscriptions and cluster bounding boxes) are matched at common overlay nodes given the SFC mapping.

4.1.4 Algorithm Analysis

Correctness

As we already mentioned in Section 4.1.2, clustering is an underdetermined problem for which no strict correctness criteria can be applied. It was also shown in the same section that, when given the region size obtained by equation 4.1, the algorithm can correctly apply the cluster definition from Section 4.1.1. There are, however, non-functional correcteness criteria that can be demonstrated.

First, we can show that the algorithm is guaranteed to terminate after the analysis/consolidation phase. It is evident that the procedure terminates because for each broadcast a node either does not receive new remote clusters, in which case it terminates, or it merges a new cluster with existing clusters to create a larger region for the next iteration. Because a region cannot grow indefinitely to include responses from new nodes, then the process must eventually terminate.

It can also be shown that the algorithm is very likely to find a single centroid for

an arbitrary cluster of uniform density (the case of non-uniform density is similar, but is slightly more complex because the conclusions do not apply symmetrically to all nodes). We have shown that we can calculate an extended cluster width that can be made increasingly likely to query neighboring nodes if a cluster extends across region boundaries. Thus, after each node has received a first round of broadcasts from its neighbors, the new regions will expand to include the regions of these neighbors. Because this information is broadcast to all of these neighbors in turn, the new broadcast will extend into these new regions (by a consideration similar as above) if the cluster extends beyond these neighbors. All of the nodes in this region will therefore receive the merged cluster info. If this process stops with more than one centroid, then it is because one or more of the broadcasts failed to cross the region boundary for their given nodes. But then the separation of the points between those regions is more than the extension of the cluster width, which is reason to consider two separate sub-clusters.

Complexity

Because most of the complexity of the algorithm depends on messaging and the mapping between nodes and regions in the information space, the complexity analysis must be done with respect to the particular overlay on which the algorithm is implemented. In this case, we use the Chord overlay and SFC mapping described in Section 3. For this analysis, n is the number of nodes in the network and N is the size of the data set, a is the number of dimensions (attributes) in the information space, and b is the number of bits used to encode values along each dimension.

Before running the algorithm, joining Chord requires $O(\log^2 n)$ number of messages, for each node, and a time that is proportional to this number. Calculating the size of the region R takes one query to the predecessor node, which takes $O(1)$ messages in Chord. Now, the data collection and analysis phases are considered. Each data point sent over the network requires $O(\log n)$ routing hops in Chord. This translates to $O(N \log n)$ messages or $O(n \log n)$ messages if $N \in O(n)$ (as in a self-monitoring network). Temporally, the complexity due to data distribution is $O(\log n)$ given that nodes exchange data points in parallel.

For the consolidation phase, we must give a bound on the number of broadcasts, as well as the messaging complexity of each broadcast. Let $m = \frac{size(C)}{size(R)}$ be the number of processing node regions occupied by a cluster C with an analysis region size R. If the cluster is in a single region, then the number of broadcasts is trivially $O(1)$. Otherwise, each broadcast will reach at least the number of neighbors of the broadcasting node. This is a constant given by $g = 3^a - 1$. Since each of these nodes subsequently makes a new broadcast until the cluster is covered, the number of broadcast rounds needed to cover the cluster is $O(\log_g m)$. Each broadcast is sent as a range query in Squid, which in the worst case takes $O(b \log n)$. Since $m \in O(n)$, the complexity of consolidation is $O(b \log^2 n)$.

4.2 Comparative Analysis

This section will present an evaluation of the decentralized clustering algorithm from accuracy, performance, and robustness perspectives. It is difficult to evaluate accuracy in the case of clustering, because there is no clearly established criterion to judge the quality or correctness of a clustering result. Different algorithms are based on different definitions of what a cluster is, and in some cases there is no single best clustering even from a single definition. Instead, the perceived accuracy of a clustering algorithm depends on the cluster definition used and the representation of the data, and is often evaluated in a purely visual way.

The approach for accuracy evaluation used here is to generate data with a known distribution (classification of points as belonging to particular clusters or to random noise) and to compare the classifications produced by both known and widely used clustering algorithms and DOC. Because DOC is designed as a density-based clustering algorithm, we use two common density-based centralized algorithms for comparison, DBSCAN and Clique, which were introduced in Section 2.2.3. The reason we use centralized algorithms as the basis for comparing accuracy is that they can perform more sophisticated analysis on the data and are thus expected to be more accurate. The purpose of the evaluation is to show that DOC is close to these centralized algorithms in terms of accuracy, making it a good tradeoff for other advantages in performance

and/or robustness.

Because DBSCAN continues to be one of the most prevalent general-purpose density-based clustering algorithms in the literature, we consider it is well-suited as a benchmark for accuracy. Clique was chosen because of its similarity with DOC (in fact, DOC is a particular decentralized version of Clique), because of which it can clearly demonstrate the effect of DOC's decentralization approach on accuracy. Both algorithms were implemented in Java according to their functional specification, using the same data representation used for DOC.

4.2.1 Accuracy Evaluation

As was mentioned earlier, the approach for evaluating the accuracy of the clustering algorithms is to compare their output to a known classification of data points into clusters and noise. This suggests using metrics that measure the errors due to misclassification of points. Furthermore, we evaluate two distinct capabilities of the algorithm that address two different functional requirements for management applications. The first is outlier detection, which can be evaluated in terms of false positive and negative rates. The second is the capability of identifying all cluster points and of distinguishing points in separate clusters from each other, for which we use two separate metrics: recall and precision, respectively. In order to calculate these metrics, we process the results produced by the algorithms as follows:

When the data points are generated as input, they are given a different label according to their corresponding cluster (points generated as noise are also labeled accordingly). Let the label of point p be denoted as $l(p) \in 1, ..., n$, where n is the original number of clusters (including noise). Let the clusters identified by a clustering algorithm be denoted as c_j, where $j \in 1, ..., m$. Since clusters c_j can contain points of different labels, we count the number of points of each label in each cluster. This is:

$$count_i(c_j) = \Big|\{p | p \in c_j \wedge l(p) = i\}\Big|$$

The label of a cluster ($l(c_j)$) is that corresponding to:

$$\arg \max_{i \in 1,\ldots,n} (count_i(c_j))$$

Finally, the representative of a label i is the cluster c_j given by:

$$rep_i = \arg \max_{j \in 1,\ldots,m} (count_i(c_j))$$

Precision is calculated as the sum of the number of points of the representative clusters over the total number of points (T):

$$precision = \frac{\sum_{i=1}^{n} count_i(rep_i)}{T}$$

According to this definition, precision will decrease both if a single generated cluster is split into more than one piece and if two or more of these clusters are merged into a single cluster result. On the other hand, recall is calculated for each cluster label i, where T_i is the total number of points with label i, as:

$$recall_i = \frac{\sum_{\{j|l(c_j)=i\}} count_i(c_j)}{T_i}$$

For recall, therefore, it only matters that the points are identified as cluster points. If o is the "cluster" of outliers (noise), then the false negative rate can be obtained from $1 - recall_o$. Similarly, the false positive rate corresponds to:

$$1 - \frac{\sum_{i \neq o} T_i \cdot recall_i}{\sum_{i \neq o} T_i}$$

We conducted a set of experiments to evaluate the impact on precision and recall of the number of dimensions, the number of points, and the point distribution (number and shape of clusters as well as percentage of noise). The choices for the different experimentation parameters are explained below:

- Number of dimensions: We used spaces from 2 to 6 dimensions. Two and three dimensional spaces are used for illustration purposes, since the quality of clustering can be ascertained visually. DOC can be used with 5 and 6 dimensions, but for these the time taken by our DBSCAN implementation became prohibitive and

could not be used for comparison. Beyond six dimensions, the clustering properties of the SFC used for point distribution start to become less effective [55] and would therefore not be as useful.

- Number of points: We clustered data sets of the order of 10^3, 10^4, and 10^5 points. Illustrative tests in two and three dimensions were conducted with 10^3 points, for ease of visualization. The remaining are the ranges expected in the targetted applications, and are used with experiments for three or more dimensions.

- Shape of clusters: Convex and non-convex shapes were used for the illustrative tests, given that the density-based algorithms should identify all types of shapes. To facilitate experimentation and because the effect of cluster shape on accuracy is beyond the scope of this thesis, convex-shaped clusters were used for all other tests. However, these include lines and elongated shapes that would trouble non-density-based algorithms.

- Noise percentage: Every experiment was conducted with high (5%) and low (0.5%) noise percentage

Illustrative Tests

The illustrative tests were done with the purpose of verifying the accuracy of the algorithms visually. Figures 4.14, 4.15, and 4.16 are the best and worst results obtained for the three algorithms (determined by the combination of precision and recall, not visual inspection) for the 2D and 3D illustrative tests.

Although DBSCAN had the best results visually, DOC had similarly good results, especially in the 2D case. Although the higher dimension cases are not shown, it is worth pointing out that DOC did better than dbscan in higher dimensions for the default configurations because of the inaccuracy of DBSCAN's parameter setting mechanism in these dimensions (see Section 4.2.1). The Clique algorithm was highly sensitive to parameter variations, as can be seen in the worst results.

The 3D test was designed as a very difficult set to cluster, because of the wide differences in cluster densities and close proximities of the clusters. This was the test in

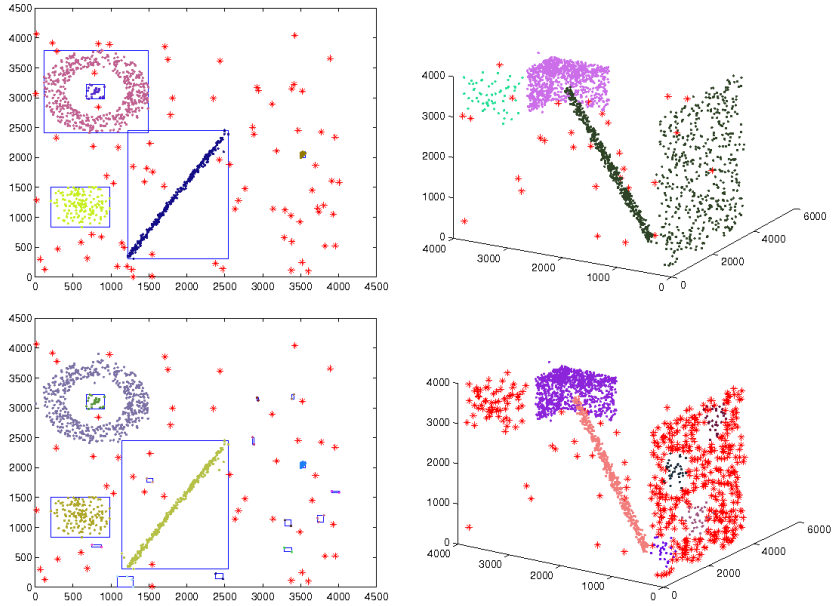

Figure 4.14: Best (top) and worst (bottom) results for DBSCAN illustrative tests for two and three dimensions

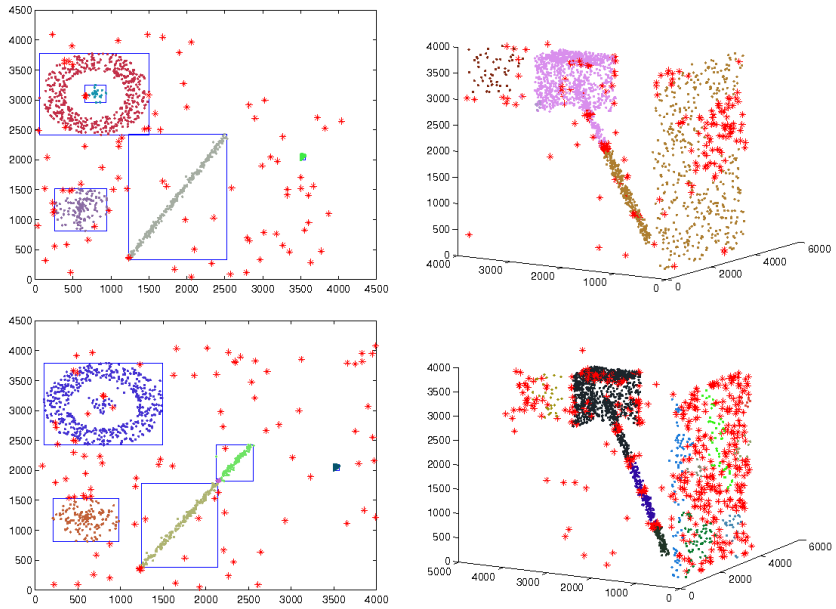

Figure 4.15: Best (top) and worst (bottom) results for DOC illustrative tests for two and three dimensions

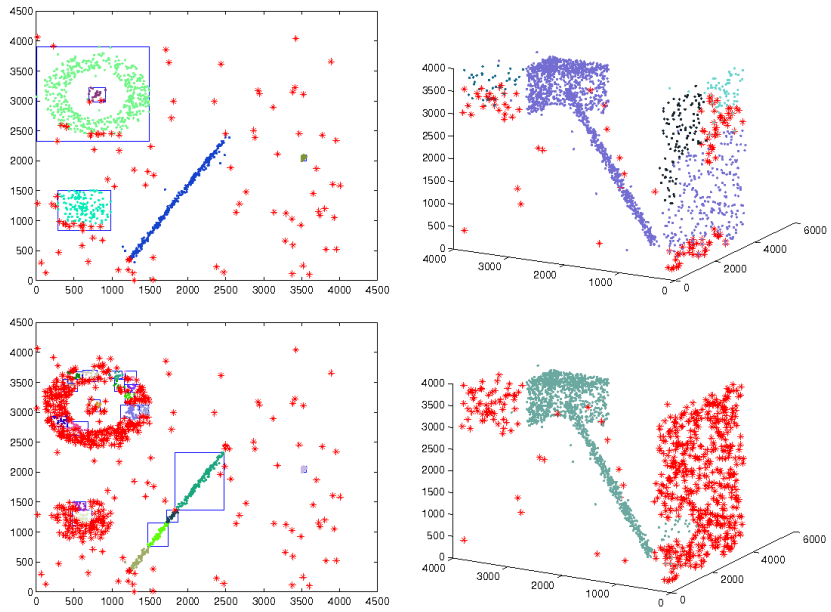

Figure 4.16: Best (top) and worst (bottom) results for Clique illustrative tests for two and three dimensions

which all algorithms performed the worst, but it is evident when observing the results that DBSCAN had the advantage because of its density connectivity analysis.

Evaluation Results

In the results that follow, the following test name conventions are used:

<Test type><Num dimensions>D[<Point exponent>X<Noise density>]

Where:

- Test type is IT for illustrative tests and ACC for accuracy
- Point exponent is 4 or 5 (for 10^4 and 10^5 points respectively). Exponent 3 is assumed for illustrative tests and is omitted.
- Noise density is L for low density and H for high density.

Figure 4.17 is the comparison of the precision of the three algorithms obtained for their respective default parameter settings, and Figure 4.18 shows the best precision obtained by the different algorithms with some parameter setting. While the best

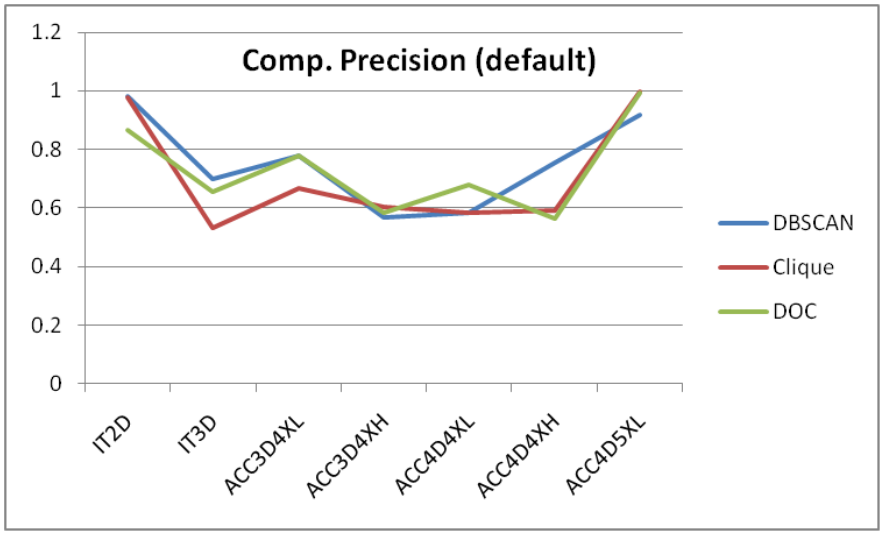

Figure 4.17: Comparison of precision results obtained from running algorithms with default parameters

results for DBSCAN were consistently obtained with a particular set of parameters (different from the default parameters), there was no such consistency in the results for Clique. The best results for DOC were often obtained with the default parameters. Another aspect to note is that the algorithms perform similarly and the quality of the result depends more on the test type than on the algorithm used. However, DBSCAN was the most consistent and accurate of the three algorithms in terms of precision.

The same comparison of the three algorithms is made in terms of average recall (the recall of all clusters in the tests over the total number of clusters) and is shown in figures 4.19 and 4.20. There is not as much variation in terms of recall as there is for precision, since all of the tests except for one had over 85% average recall. When discriminating the recall of different clusters (including noise), it is noise that consistently has the lowest recall for all of the algorithms, as can be seen from the false negative and positive rates.

When comparing the algorithms with respect to these outlier detection error rates, it is evident that the false positive rate is consistently low or zero for all algorithms, with no evident trends in the data. However, false negative rates are higher for all of the algorithms. Figures 4.21 and 4.22 show the comparison of the false negative values across the different tests. With default parameters, DOC performs consistently at an

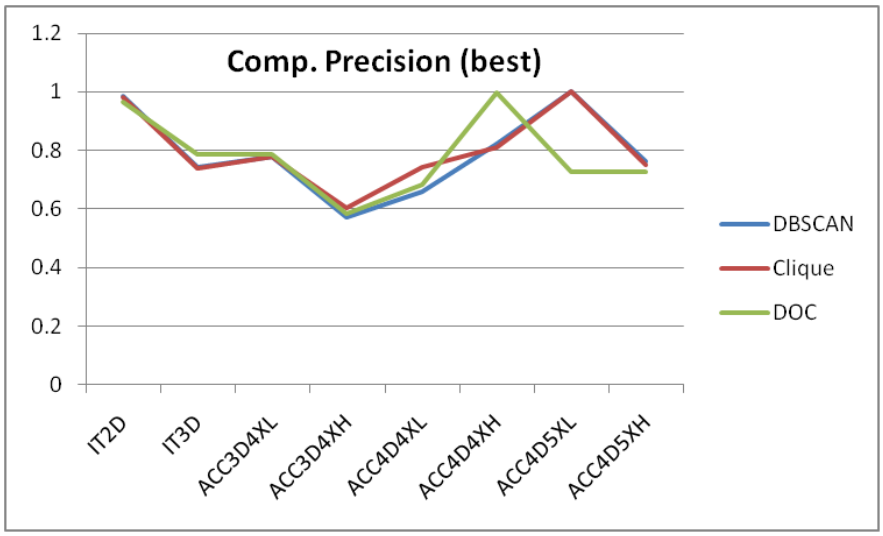

Figure 4.18: Comparison of precision results obtained from running algorithms with best parameters

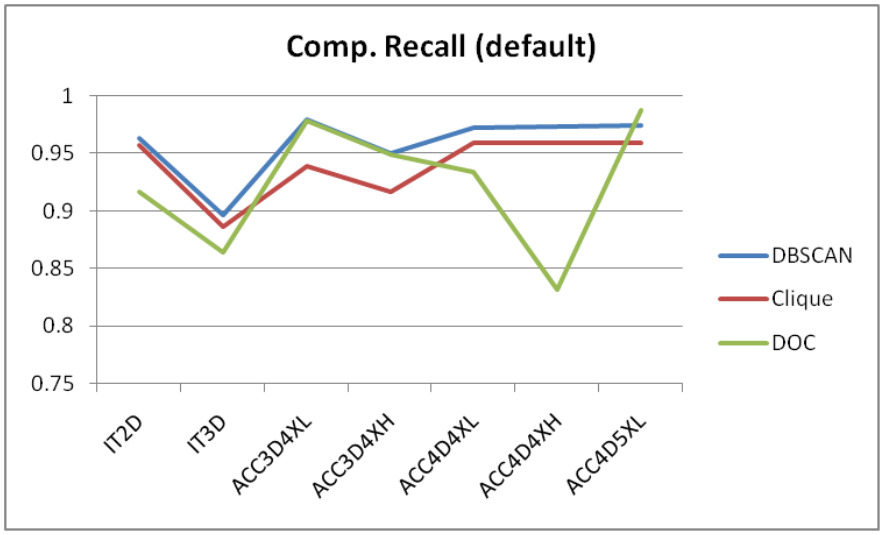

Figure 4.19: Comparison of recall results obtained from running algorithms with default parameters

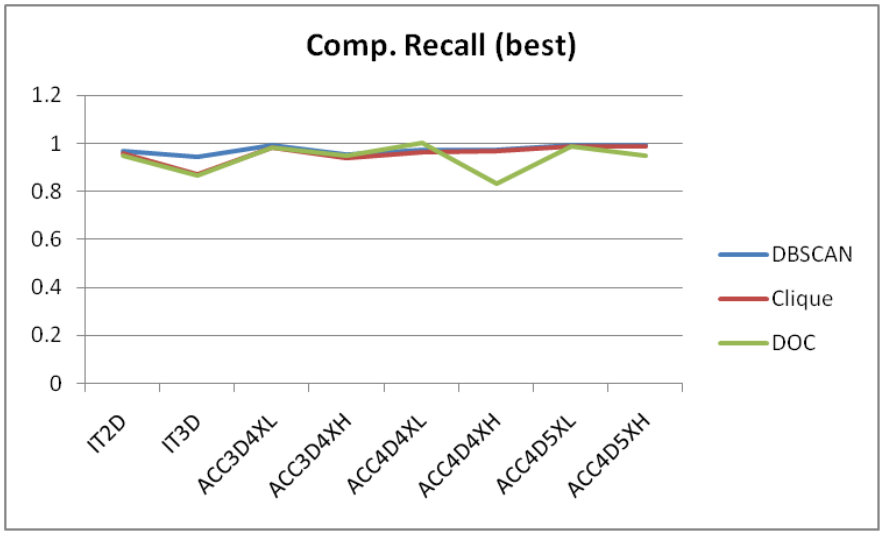

Figure 4.20: Comparison of recall results obtained from running algorithms with best parameters

average false negative rate of about 15%. Unlike previous cases, DOC does better than DBSCAN and Clique for smaller data sets, possibly because of lower data densities in the noise. It is clear from 4.22 that both DBSCAN and Clique are able to obtain better results when provided with the correct parameters, but these are generally not the parameters that can be found using the pre-processing method.

The relatively high false negative rates can be explained by the fact that noise is generated randomly in the entire space, and thus a fraction of noise points inevitably falls within existing clusters. To verify this, we calculated a metric called the local outlying factor [14] (LOF) for the missed noise points. The LOF value is expected to be close to 1 for points that are within a cluster and higher the more isolated a point is from clusters in the space. Figure 4.23 shows the distribution of LOF values for all of the different runs. For all algorithms, the majority of values are indeed close to one within the range of values obtained, which indicates that the points being missed are those that are close to or within a cluster.

Sensitivity Analysis

The sensitivity of the three clustering algorithms was evaluated with respect to three basic aspects: 1) the characteristics of the information space, 2) the cluster distribution,

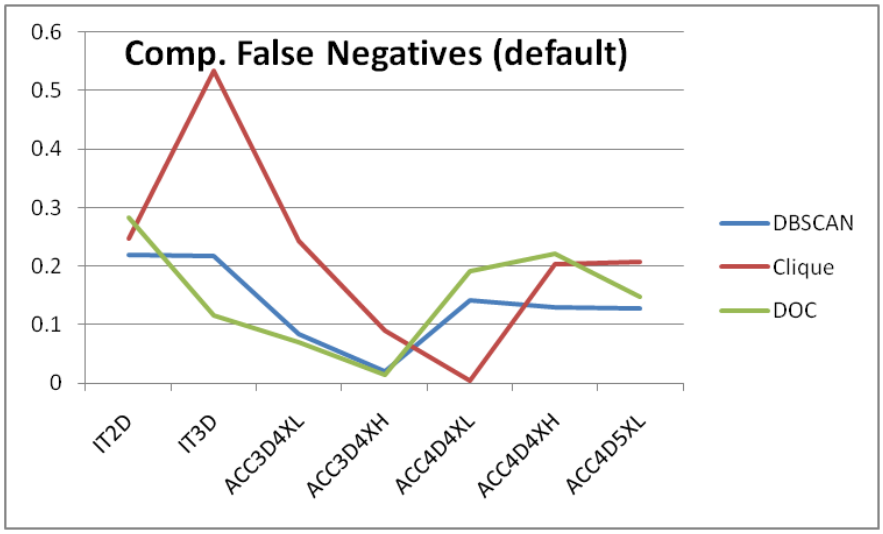

Figure 4.21: Comparison of false negative results obtained from running algorithms with default parameters

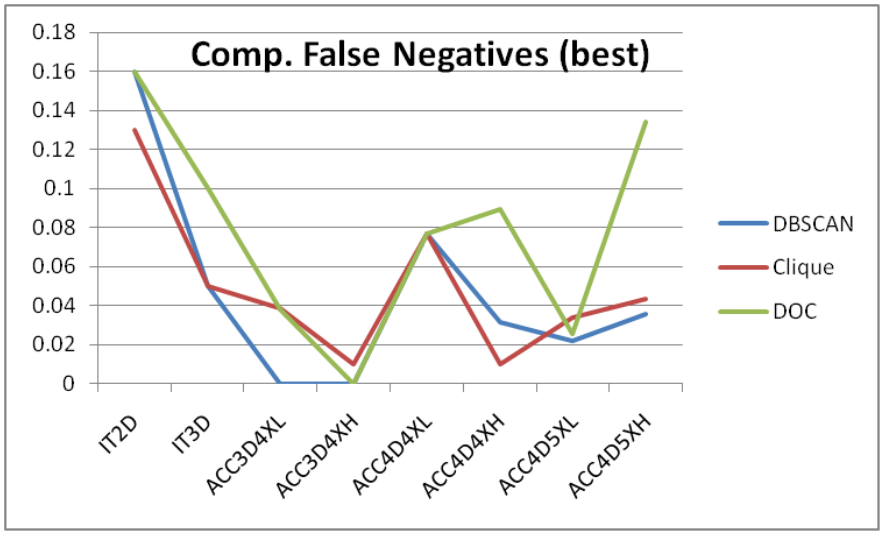

Figure 4.22: Comparison of false negative results obtained from running algorithms with best parameters

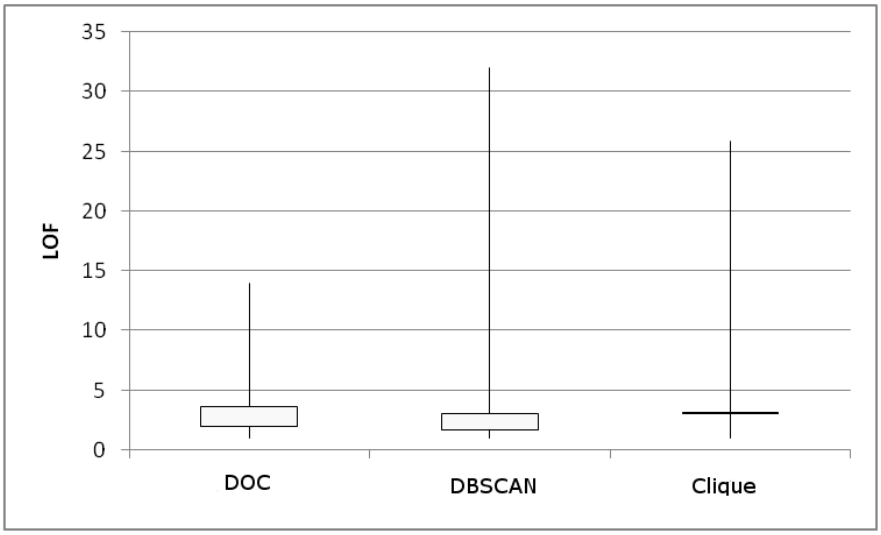

Figure 4.23: Box and wisker plot of LOF values of false negatives. The values within the box are between the mean and median LOF values for the outliers missed by each algorithm.

and 3) the individual algorithm parameters. Information space characteristics are the number of dimensions, the number of points being clustered, and the percentage of noise in the data. The sensitivity of the algorithms to these characteristics shows inherent strengths and limitations for their application to particular contexts. By contrast, the cluster distribution is a dynamic runtime aspect that includes the number, size, location, and shape of the clusters. Although it is difficult to exhaustively and systematically test the effect of different cluster distributions on the accuracy results, this was not really the purpose of our analysis. Rather, we sought to establish whether or not there was any effect due to distribution, in order to contrast this with the effect of other aspects. The parameters of each algorithm affect the density threshold used to detect clusters and outliers. The more sensitive an algorithm is to the setting of its analysis parameters, the better an initial method or procedure for finding the best parameter values has to be. This means that a very sensitive algorithm may not be suitable for autonomous operation, unless a sufficiently intelligent agent is available to set the parameters to achieve the best accuracy.

For each of the separate characteristics of the information space, we calculated the

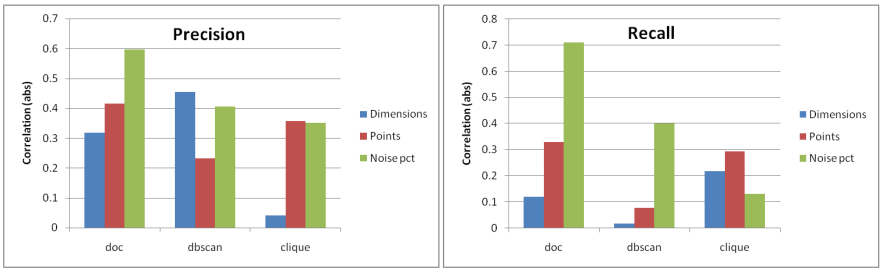

Figure 4.24: Correlations of information space characteristics with algorithm accuracy. The quantities shown are absolute values since some of the correlations are negative.

correlation with both accuracy metrics (precision and recall). Each of the characteristics was varied for the different types of tests, which are enumerated in Section 4.2.1. Figure 4.24 shows the absolute values of the correlations found for each algorithm. For the most part, there are no significant correlations, particularly for the number of dimensions and number of points. The accuracy of DOC, however, does appear to be affected directly by the noise percentage (being that with recall the largest correlation value found). The reason that recall is affected in this way by noise percentage is explained by the analysis of the false positive rate of the previous section. However, DOC recall may be more sensitive that the other algorithms to the increase in noise (though DBSCAN is also sensitive to it to some extent) because the density threshold for it is not set manually, with the data distribution in mind. Although it may be concluded that DOC is better suited to environments with a low incidence of outliers, which is indeed the case of the applications studied in Chapter 5, note that the lowest average recall for DOC with its default configuration is 81% (see Figure 4.19).

The aspect that affects accuracy the most is cluster distribution (point 2 above). In order to evaluate the effect of cluster distribution without exhaustive exploration, every test configuration was repeated for three different distributions. This means that, across test repetitions, the location, size, and shape of the clusters were generated randomly. However, the different algorithms were all run on the same test variations to ensure a consistent comparison. As can be seen in Figure 4.25, the normalized accuracy differences between the different distributions on each of the test types were nearly identical for all of the algorithms. This means that the "difficulty" of the distribution determined the accuracy of the algorithms more than any other factor. Furthermore,

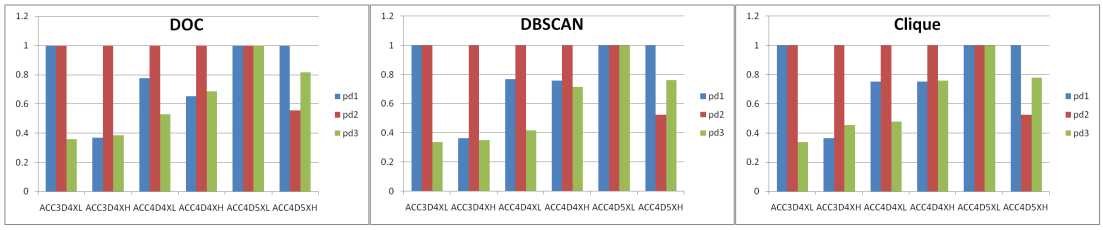

Figure 4.25: Effect of cluster distributions on algorithm accuracy. Each subfigure corresponds to a different algorithm and shows the different test types and cluster distributions (pd1 – pd3).

the fact that the effect was the same for all of the algorithms means that DOC is not inferior to the others in its ability to handle the different cluster distributions.

The final sensitivity evaluation is with respect to the algorithm parameters. The parameters for DBSCAN are the minimum number of points ($MinPts$) in a point's neighborhood and the size of the neighborhood (EPS). The parameters for Clique are the number of divisions of the range of each dimension of the space ($Divisions$), and the percentage of the total number of points ($PointPct$) needed to identify a cluster in any division. DOC uses the region size as a parameter, obtained from a value of $MinPts$ as described in Section 4.1.2. DBSCAN provides a visual method for parameter estimation, that could conceivably be automated, but with pre-processing of the data. The literature found for Clique does not provide a mechanism to determine its configuration parameters, but since they are roughly equivalent to those of DBSCAN, the same set of parameters were used for each. Each parameter was varied independently by 25% and 50% below and 50% and 100% above the default values. For DBSCAN and Clique, when one parameter was varied, the other was left at its default value.

Using the accuracy results from the runs with default parameters as a reference, we obtained the difference in both precision and recall with each of the tests run with the modified parameters. To better see the range of effects of the parameter changes, we obtained distributions of the differences and plotted them as histograms in Figures 4.26, 4.27, and 4.28, for DBSCAN, Clique, and DOC respectively. There are two histograms for each algorithm parameter, one for precision and the other for recall.

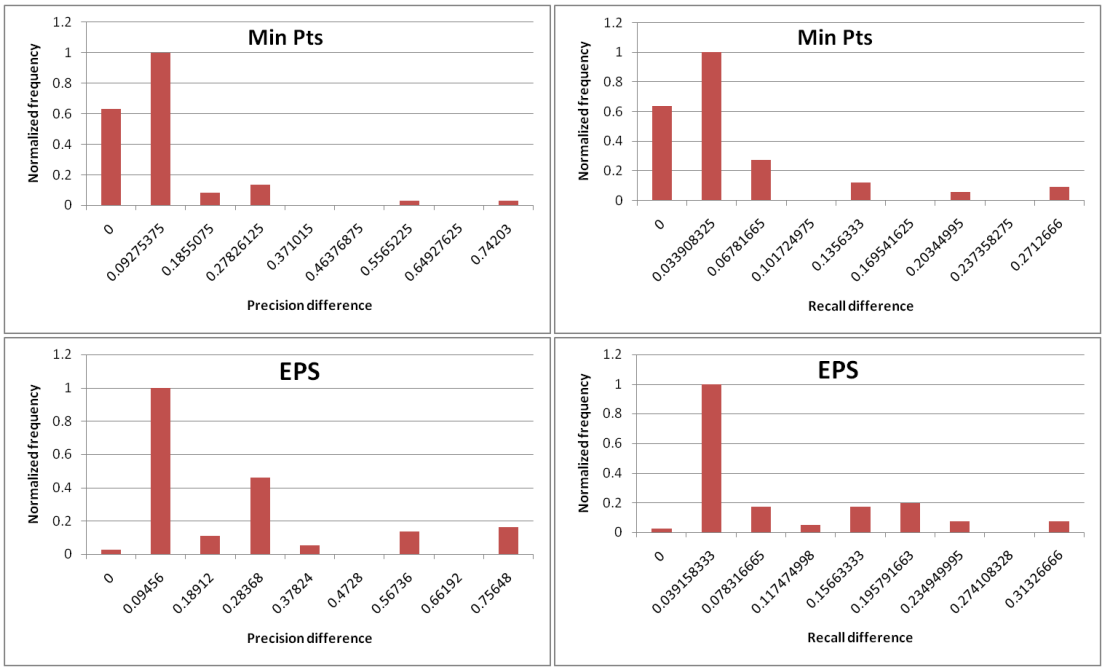

Figure 4.26: Histograms showing the distribution of effects of DBSCAN parameters on precision and recall. The values shown are relative frequencies of results that differ by the given value from the results with default parameters.

For DBSCAN, the differences due to the variation of $MinPts$ are small (most between 0 and 9 percentage points[2] and 0 and 3 percentage points for recall). There are some extreme cases, which are attributable mostly to specific cluster distributions. For EPS, however, there is a broader distribution, with 40% of the maximum between 19 and 28 precision points. Recall is also relatively insensitive to variations in EPS, although the distribution has values of up to 20% of the maximum between 15 and 20 points.

Clique, on the other hand, is very sensitive to parameter changes, especially to changes in the number of divisions. Most precision results are between 12 and 37 points, and a significant amount are between 50 and 63 points. Although the other distributions are not as marked, Clique is clearly the most sensitive algorithm.

In terms of precision, DOC is comparable to DBSCAN, with most results between 4 and 12 points. Like the other algorithms, it is also insensitive in terms of recall, with

[2]so called because they correspond to the metric unit, which is a percentage, and not to a percentage of the default value for the metric

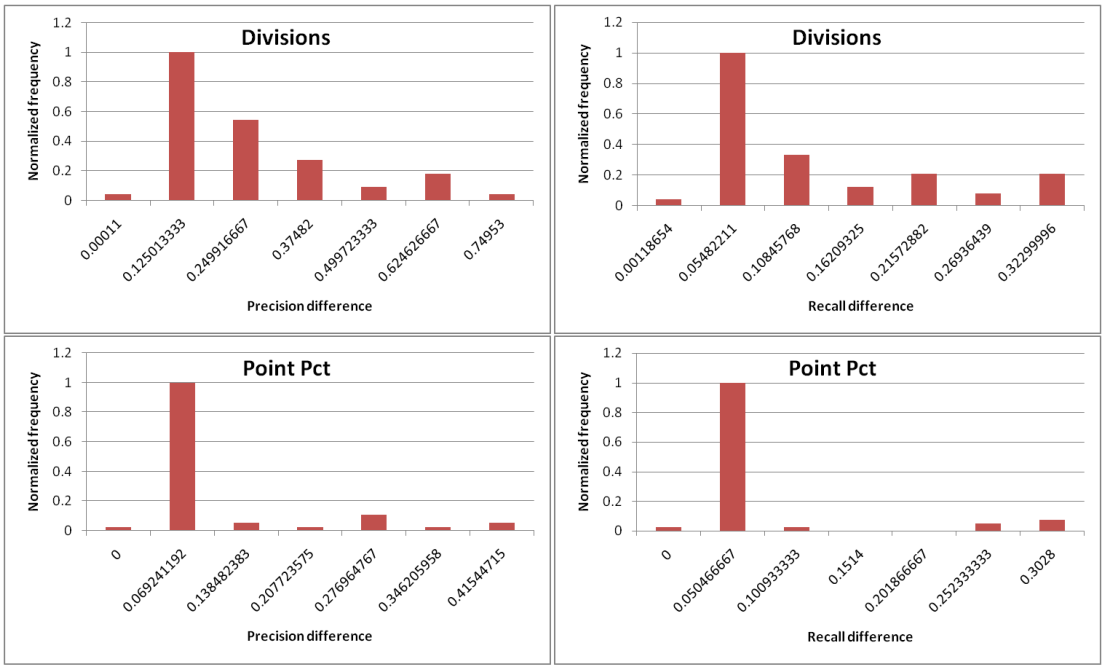

Figure 4.27: Histograms showing the distribution of effects of Clique parameters on precision and recall.

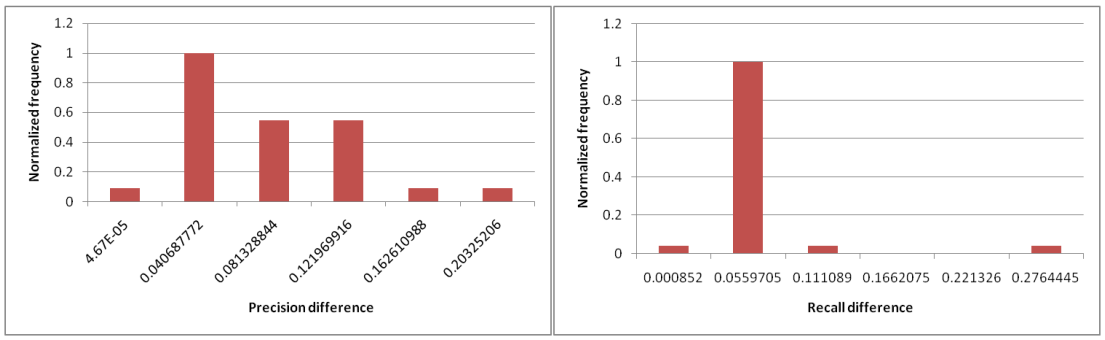

Figure 4.28: Histograms showing the distribution of effects of DOC parameters on precision and recall.

most results falling within 5 percentage points. Unlike the other algorithms, however, the extreme values for the distributions are not as large, for either precision (up to 20 points vs 75 for both DBSCAN and Clique) or recall (28 points vs 30-32).

We mentioned earlier that the more insensitive an algorithm is to its parameters, the more suitable it is for automated execution in a dynamic environment, because it relies less on the accuracy of setting those parameters. These sensitivity results show that DOC is well suited for automated, online execution using the parameter calculation explained in Section 4.1.2, while both DBSCAN and Clique are less so. This is not only because of the distributions above, but also because of the observation that these algorithms often did not obtain their best results with the default parameters, especially for higher numbers of points and dimensions, unlike the case of DOC. We further sustain that the reason for the relative insensitivity of DOC is that the initial calculation of the density threshold is based on a uniform distribution of points and not on the region size parameter. Although cluster detection ultimately does depend on region size, because the minimum number of points is obtained from the density and the region size, it is a weaker dependency than that of the other algorithms.

4.2.2 Performance Evaluation

Algorithm running times

Online execution of clustering analysis depends significantly on the possibility of running the algorithms on the necessary data and obtaining the results within a reasonably short time frame. This allows the analysis to be done within the same time window of data generation. Analyzing the running times of DOC and its centralized counterparts is thus important to understand when decentralization is necessary or at least a good tradeoff with respect to accuracy. There are two components in the running time when the algorithms are applied in a distributed system. In this section, we consider the running times due exclusively to data analysis and result aggregation (in the case of DOC), obtained from the experiments performed in the accuracy evaluation. The time due to the distribution of data over the network will be analyzed in the next section.

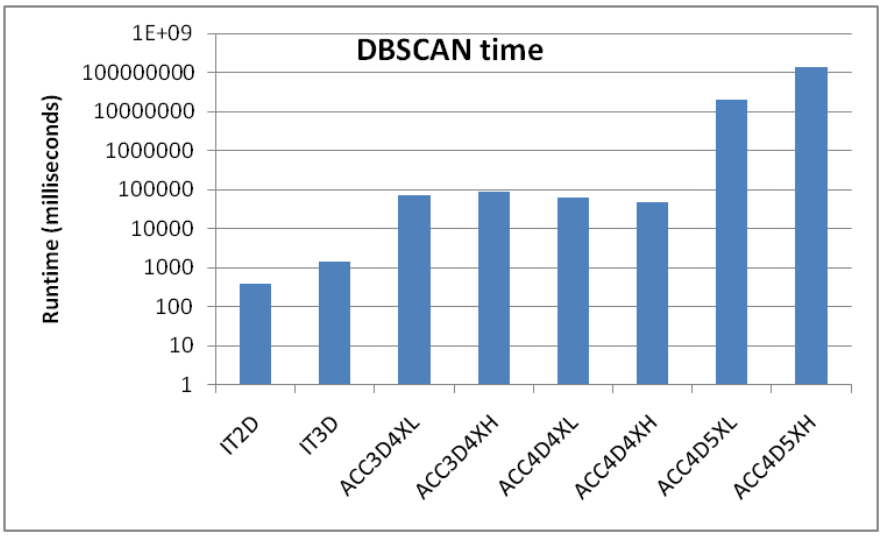

Figure 4.29: DBSCAN running times

The first observation that can be made is that the running times of DBSCAN show poor scalability, as can be seen in Figure 4.29. Notice the logarithmic scale on the time axis (measured in milliseconds). It should be noted that the implementation of DBSCAN used for these experiments is not optimized, and significant improvements in the running time of the algorithm can be obtained with appropriate optimizations. However, the worst case complexity of the algorithm and the results presented in [26] still suggest the same growth trend.

Until the order of 10^4 points, the performance of DBSCAN is good, taking slightly over one minute in the worst case. For the order of 10^5 points, however, the duration of the experiments was in the order of hours, and it was affected significantly by the number of dimensions and the algorithm parameters. Note in the figure that ACC4D5XL and ACC4D5XH, which have the same number of points, but different noise percentage, took $5\frac{1}{2}$ and almost 39 hours, respectively. Because of these running times, it was prohibitive to complete the rest of the experiments in higher dimensions.

In contrast, Clique is a very fast algorithm, as the time chart in Figure 4.30 shows. All of the times obtained are less than one second. Because DOC is essentially a decentralized version of Clique, the results of Figure 4.30 suggest the potential of the decentralized implementation. The overhead of decentralization, however, is significant when compared to the centralized result. In Figure 4.31, we can see the favorable

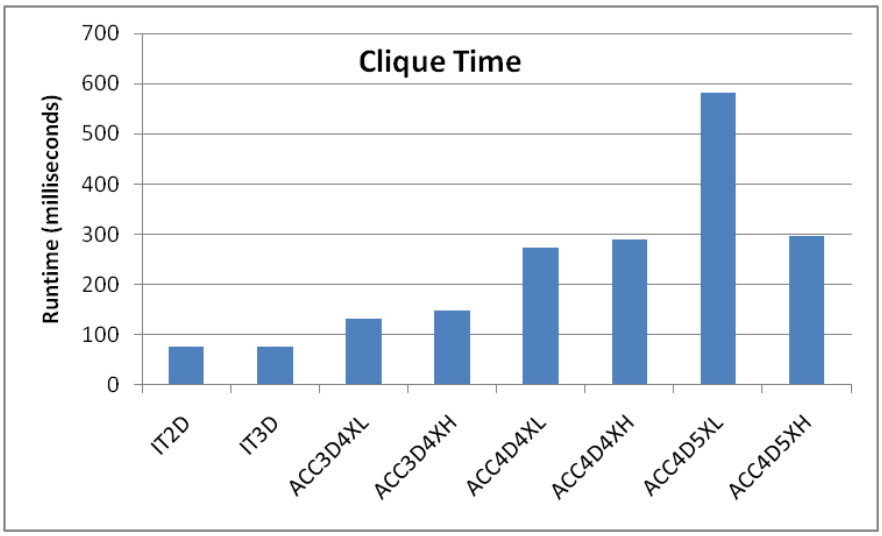

Figure 4.30: Clique running times

scalability of the runtimes obtained with DOC, but also their increment with respect to Clique. There are two main reasons for this. The first and most obvious reason is the messaging overhead of the rounds of communication necessary to aggregate the results across processing nodes, as described in Section 4.1.2. However, what appears to be the most significant reason is that, due to the loosely-coupled communication infrastructure, it is difficult for nodes to determine when no further communication from other nodes is forthcoming and therefore when its results are complete. In the current implementation, a node implements a timeout that is refreshed every time an aggregation message is received. Only when the timeout expires does the node decide that its result is complete. Clearly, the runtime is highly dependent on this timeout value.

Despite the high runtimes with respect to Clique, DOC performs well, within two minutes for all experiments of up to six dimensions and 10^5 points (Figures 4.31 and 4.32; the higher dimensionality tests are shown separately for DOC because they were not replicated with DBSCAN and Clique, due to the excessive runtimes of the former). The times scale well and depend more upon the data distribution than on the number of points, since some experiments with 10^4 points took longer than others with 10^5 points. When compared to DBSCAN, the decentralized implementation becomes favorable after 10^4 points. Also, recall that the results seen so far discard the point

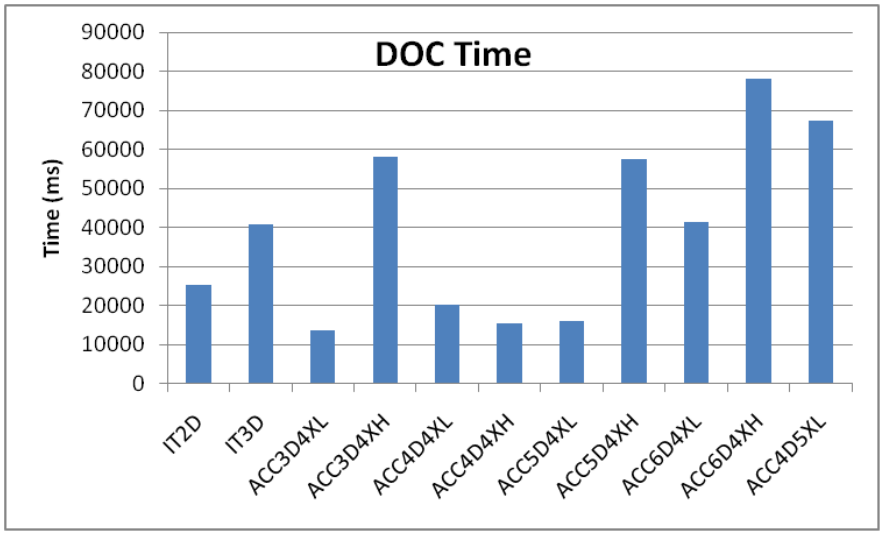

Figure 4.31: DOC running times

distribution time, i.e. the time taken to transmit points over the network to processing nodes. If the data is initially centralized, the distribution time gives an advantage to centralized algorithms. However, if data is initially distributed, as is the case in distributed systems and the applications we consider, centralized algorithms also incur overhead due to the initial centralization of the data. In the next section, we analyze this overhead with respect to the redistribution of data in DOC and further determine the circumstances in which the decentralized implementation is advantageous.

Figure 4.32 also includes tests run on the Amazon Elastic Compute Cloud [1], a virtualized computing infrastructure where virtual machine instances can be configured and created on demand. For these tests, a large file with over two million points was divided among the nodes that ran the clustering algorithm. The first test was run with 50 nodes and one million points, and the second with 100 nodes and the full two million points. These tests show that the scalability of the approach can be maintained by increasing the number of nodes in the clustering network. Note, however, that The times in the figure had to be adjusted to account for a delay in point generation introduced to avoid blocking of the node network, a problem which we attribute to either network bottlenecks on Amazon's infrastructure or load imbalance (an issue discussed in Section 4.1.2 and then again in Section 6.1).

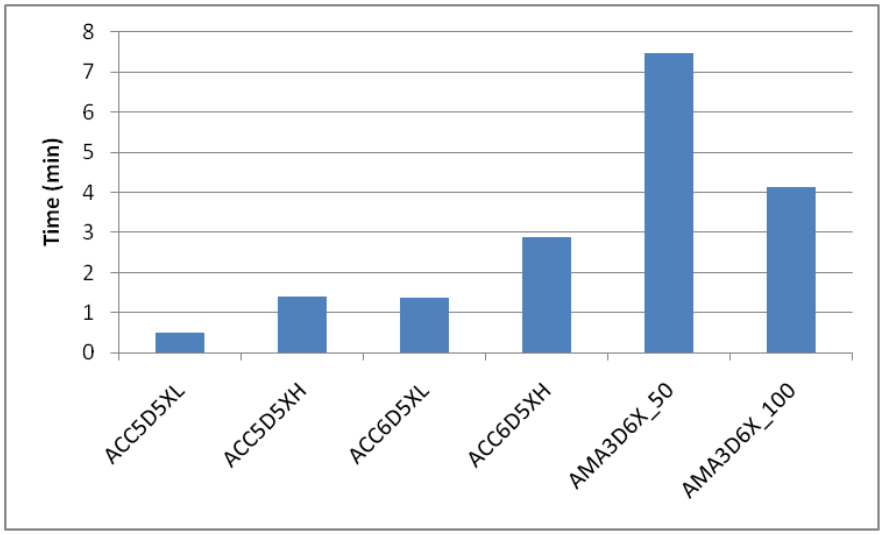

Figure 4.32: DOC running times for higher dimensions, and including Amazon EC2 tests

Data Distribution Analysis

The experiments above were conducted with data generated from a single source and distributed to processing nodes in the case of DOC. However, in an online management scenario with monitoring data being produced by system components in real time, data is already distributed. It makes intuitive sense, then, that processing the data in-network with a distributed set of nodes would be more efficient than aggregating the data in a centralized location for processing. This must be verified empirically, however, because DOC requires a re-distribution of data to nodes in which every point goes through $O(\log n)$ routing hops. A distributed algorithm like DOC will therefore be preferable (in terms of distribution overhead) to a centralized algorithm if the latency of routing is compensated by a reduction in synchronization overhead due to spreading out the messaging load.

We assume that online clustering is done on data that is produced from distributed sources at periodic (though not necessarily uniform) intervals. The application of DOC, as described in Section 4.1.2 consists of two alternating or overlapping phases that repeat over time: a data collection phase, in which nodes listen for the data routed to them via the overlay from the distributed sources, and an analysis phase, in which clustering results are produced. In order for online clustering to be effective, the analysis

phase must, at most, complete within the same time period as that of the data collection phase. Otherwise, the analysis phase would overrun the data collection phase and the online process would not be sustainable.

The above means that the time added by point distribution to the execution of the algorithm should be minimal. Ideally, point distribution should completely fit into the time allotted for the data collection phase. Therefore, in order to compare centralization against the redistribution of points in the overlay, we measure the *overrun* time: the amount of time for which the clustering application is still receiving points after all points have been sent. As another point of comparison, we also show the difference in average bandwidth consumption during the distribution time for the overlay nodes and the central node.

The experimental setup is as follows. In our 16 node cluster, sets of 10^4 and 10^5 points are divided equally among the nodes, which form a Squid/Chord overlay (see Section 3). The points follow an artificially generated distribution, as in the accuracy experiments, with a single cluster and differing levels of noise. The reason for this is to simulate the worst case for DOC distribution in which most points are routed to a reduced set of nodes. Points are generated by the nodes with a Poisson distribution, and we conducted three sets of experiments, each with a different point generation rate. These rates were 1, 2, and 10 points per second. For each experiment, the analysis window (i.e. the duration of the data collection phase) was made to be equivalent to the number of points per node times the data generation period. This way, all points will be sent by the end of the analysis window, and it is possible to measure the overrun time.

For the experiments conducted on redistribution, the nodes inserted the points via the overlay interface with the given generation period until all points were exhausted. During this time, they may also receive points that have been generated and routed to them. However, they start measuring the overrun time only when they have inserted all of their points, so that the final overrun time will be the time elapsed between the end of the analysis window and the moment the last point is received by the node. The experiments conducted on centralization were simpler, since all nodes send their

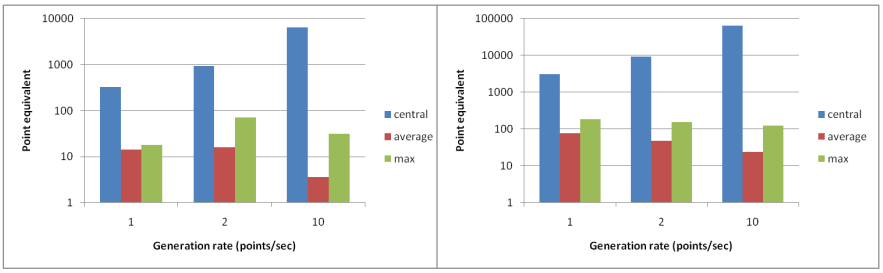

Figure 4.33: Overrun time results, in terms of point equivalents, for distribution tests conducted with 10^4 (left) and 10^5 (right) points. Notice the log scale on the Y axis.

points directly to the central node. This node keeps a timer that expires at the end of the analysis window, so that it can likewise record the time from that moment until it receives the last point.

The results of the experiments are summarized in Figure 4.33. To construct these figures, we collected the overrun times for each node in each of the experiments and obtained the average and maximum times. Since points have different generation intervals in the different sets of experiments, and in order to compare their results, we divided the total overrun times by the length of the generation period. This is roughly equivalent to a number of points received during this period, but not exactly due to network delay. The results from the overlay nodes are compared to the results at the central node receiving all of the points. In the figures, notice the logarithmic scale that shows the orders of magnitude difference between the results for centralization and redistribution. In terms of time overhead, the comparison is between a maximum of 4% of the analysis window for redistribution vs 65% for centralization for 10^4 points, and 2% vs 65% respectively for 10^5 points.

As an additional comparison between the distributed and centralized setups, we measured the application bandwidth consumption for incoming messages. Although it is logical that the bandwidth consumption will be greater at the central node, which receives the aggregated stream of points, we want to see the magnitude of this difference when compared to the proportion with the number of distributed nodes. If the messaging overhead among the distributed nodes is high, then the reduction in bandwidth will be less than proportional to the number of nodes. In the distributed case, we also

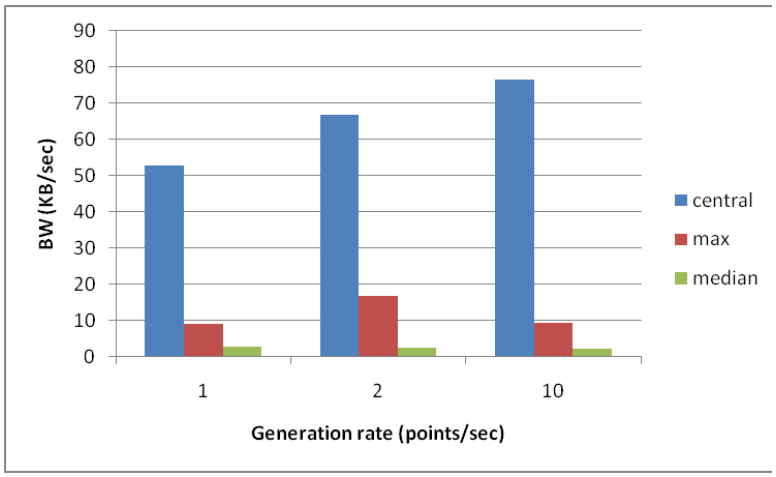

Figure 4.34: Comparison of bandwidth consumption in data distribution.

want to see if there is any significant skew in the bandwidth consumption of different nodes due to the clustering of data (recall, we use the worst case of a single cluster). In Figure 4.34, we can see the differences in bandwidth consumption for the different generation rates of the distribution experiments, comparing the centralized case to the maximum and median of the distributed nodes. Notice that there is some skew in the consumption of the distributed nodes, seen in the difference between the maximum and median values. Finally, the messaging overhead has negligible impact on the bandwidth consumption at the distributed nodes.

4.2.3 Robustness Evaluation

The evaluation of the effectiveness and robustness of the clustering algorithm is done with respect to two metrics: the error rate of outlier detection, in terms of false negatives, and the overhead of replication. As with previous accuracy experiments, the error rate is measured against a known data distribution, so that it is possible to determine whether points are labeled correctly as outliers by the algorithm. In this case, the distribution used to generate the data is based on data collected from a device network (used as test case for clustering-based management in Section 5.2.1), assuming that the points drawn from the lowest frequency intervals are anomalous and should therefore be labeled as outliers.

We use the average length of the replication chains (see Section 3.2.3) for a run of the algorithm as a metric that represents this overhead cost, since longer chains incur greater overhead. The baseline for this metric is that of full replication, because the average chain length in this case is the maximum length r^*. Below, we summarize the replication mechanism, that corresponds to the heuristic multi-node replication model proposed and analyzed in Section 3.2.3.

1. When a point i is first received by a node j, this node calculates the probability of outlier a_i based on previous algorithm results. The current length of the replication chain (up to the predecessor) for that point, given by r'_p, will be 0 at this stage and is included in the corresponding event message.

2. Node j, which has local failure probability p_j, decides whether or not to replicate the point further based on a replication probability q_j calculated as $avg(p_j, a_i) \times h(r'_p)$, where h is a decreasing probability function based on r^* that guarantees that the final r is not greater than r^*. This could be, very simply:

$$h\left(r'_p\right) = 1 - \frac{r'_p}{r^*}$$

   This function is 1 at the first node, and thus only depends on the failure and outlier probabilities, and 0 when r'_p equals r^*.

3. If the decision to replicate is affirmative, then the successor will receive the point, store it, and calculate its own q_j. Otherwise, the replication chain ends and r is given by r'_p.

In our experiments, processing node failures are independent and are modeled by Poisson processes, simulated by a series of Bernoulli trials in which each node fails with probability f at each time step. For all experiments, we have two scenarios: one with uniform failure rates and another in which more vulnerable nodes have a relatively higher failure rate. In particular, we assume an adversarial attacker that targets nodes responsible for regions likely to contain outlying points, so that f is set equal to the

outlier probability calculated at each node. Consequently, the nodes that are most likely to fail are those with a greater outlier probability. We also calculate r^* for full replication as 5, as in Section 3.2.3. This is the value we use for comparison with our experiments below.

We need to show how our selective replication mechanism produces a reduction in the error rate that is comparable to full replication, but also significantly reduces the average replication chain length. We evaluate the false negative rate of the clustering algorithm in four cases: Without failures, with failures but without replication, with failures with full replication, and with failures and selective replication. For each of the four cases, we ran the algorithm for different numbers of processing nodes, once for each of ten sets of data points, and averaged the results. In the case of random failures, we also varied the failure rate from 5 to 50%. The results for full replication are the same as those obtained without failures. Therefore, we only show the result of no failures as a baseline comparison for the two other cases.

Selective replication is set up as follows. Because of the way in which failures are simulated, we assume that each node has an accurate approximation of its own failure rate, as it would be obtained using the Bayesian update rule from Section 3.2.3, and fix this value to the corresponding rate for each trial. The outlier probability is calculated based on the Bayesian formula. The conditional distributions for point distances are taken from the distribution used to generate the data points, assuming that this history is known to each node. Selective replication experiments are conducted in two stages. A first run initializes the a-priori probabilities $P[A]$ based on the update formula, applied when a node detects outliers in its region. The results shown are from a second run of the algorithm that uses these probabilities. When applying the algorithm online, of course, the a-priori probabilities are updated continuously during the clustering runs.

Figure 4.35 shows the false negative error rates of the no replication and selective replication strategies for differing failure rates, as compared to the case of no failures. Lines with hollow markers correspond to results for no replication, while the corresponding filled markers show the results for selective replication. It is evident that selective replication significantly improves on no replication, and is close to the results

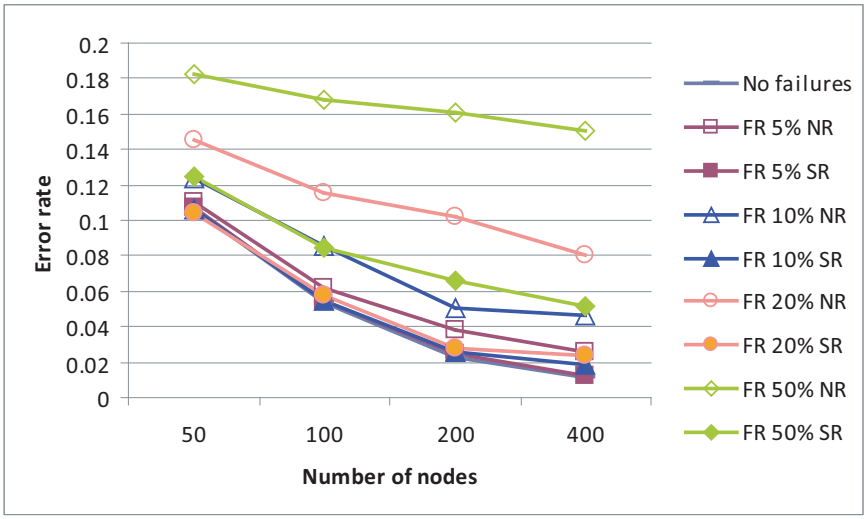

Figure 4.35: Error rates vs. number of nodes for different replication strategies and failure rates (FR=Failure rate; NR=No replication; SR=Selective replication)

Failure rate	**Nodes**			
	50	100	200	400
0.05	0.122	0.147	0.16	0.164
0.1	0.121	0.147	0.16	0.164
0.25	0.318	0.367	0.4	0.414
0.5	0.384	0.451	0.505	0.511
Targeted	0.398	0.525	0.593	0.552

Table 4.1: Average replication chain lengths based on failure and outlier probabilities

of no failures even in the face of high failure rates. Similar results for targeted failures can be seen in Figure 4.36.

The other important result is the cost of selective replication when compared to that of full replication, in terms of messaging overhead (measured as average replication chain length). Table 4.1 shows the average replication chain lengths for the different trials conducted, including targeted failures. Notice that the value is less than 1 in all cases. This is because points are not replicated at all at many nodes, so that the total replication overhead is a small fraction of the numbers of points being clustered. In contrast, full replication requires at least one message per data point, and as was calculated above, as many as five per point to provide the given guarantee that no point is lost due to failure.

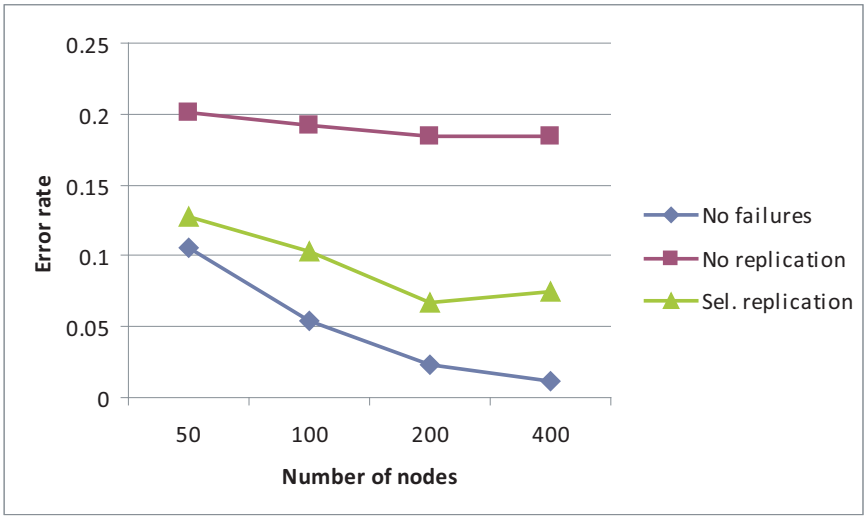

Figure 4.36: Error rates vs. number of nodes for different replication strategies and targeted failures

4.2.4 Result Analysis

The following points summarize the main conclusions that can be drawn from the experimental evaluation of DOC and the algorithms to which it has been compared in terms of accuracy and performance:

- The experiments showed that DOC is close in terms of precision and recall, including outlier detection, to the centralized algorithms DBSCAN and Clique. In addition, it is less sensitive to changes in its input parameters than the centralized algorithms, which is an important aspect for online, automated application.

- The running time of DOC scales well with the size of the data set to be analyzed, and is preferable to sophisticated algorithms like DBSCAN from the order of 10^4 points on. DOC completed all experiments in under 3 minutes, which makes it suitable for online application in several management scenarios, which will be described in Chapter 5.

- The overhead of decentralization, mostly due to cluster consolidation, was significant when comparing DOC's performance to that of Clique. On the issue of centralization vs. decentralization, it is important to consider the impact on centralizing data points that are originally distributed. The experiments showed

that this overhead is indeed significant in terms of overrunning the time of the data collection phase of the online analysis, as well as in terms of bandwidth when compared to the decentralized implementation.

- The MNR replication model used for robustness predicted that the loss probability of important points can be bounded with a short expected length of replication chains, within given bounds of node failure probability. Implementing suitable heuristics, the experiments confirmed the suitability of the model for maintaining robustness at low cost in terms of replication chain length.

Chapter 5

Decentralized Online Clustering for Autonomic Management

In this chapter, we explore how to use clustering results for decision-making, as a way to close the loop of the autonomic management cycle. Specifically, we describe the definition and dynamic application of management policies, using profiles of distributed system components obtained from the results of the clustering analysis. The mechanisms developed in this chapter are then applied to different use cases, including device network and data center resource management and usage control.

5.1 Dynamic Policy Management Framework

Policies are a powerful means of expressing high-level, goal oriented parameters that manage the behavior of systems and users, and are thus valuable tools for autonomic management. They provide a way of reducing the complexity of management tasks and allowing human administrators to focus primarily on the definition of these policies at a high level. However, polices are typically defined with static constraint thresholds and are either associated with specific states of the managed entities or applied reactively in response to feedback from events or actions. This limits their applicability to situations where the appropriate management actions depend on dynamic system properties, which require adapting policy application thresholds and parameters without modifying absolute policy definition constraints.

There is a gap that exists between goal-driven policies expressed in terms of these absolute constraints and the actual thresholds on operational parameters that must be applied so that these constraints are met. Clustering analysis can be used to characterize both the operational state of the system and the dynamic mapping that exists between this state and management goals. This is done by first identifying dynamic run-time patterns within relevant events and correlating these patterns with specific

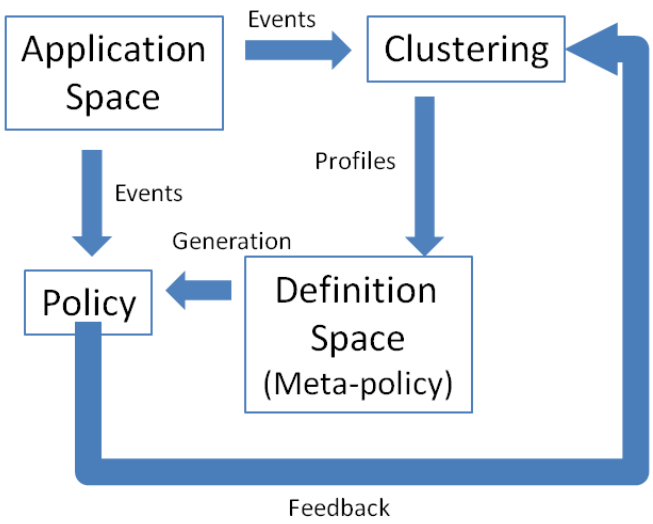

Figure 5.1: Overview of the dynamic policy approach. Note that policies are derived from meta-policies using clustering results and are applied to events in the application space before feedback is available.

states of operation for which management policies can be defined. The online clustering approach can handle the large volumes of data that can be obtained by monitoring, the distribution of this data, and the multiple attribute dimensions of the information space. Policies can then be defined in terms of these evolving patterns and associated with them with the correct events at the correct instance. We propose a scheme in which policy definitions account for these dynamics, while at the same time are easily realized in terms of absolute constraints and state descriptors.

The following sections present a conceptual framework for policy definition in terms of event-based descriptions of system state, as well as for the dynamic application of these policies based on a mapping of system states to an agglomeration of patterns in run-time events. Figure 5.1 shows an overview of our approach.

5.1.1 Definitions

Before going into a detailed description of our approach, we enumerate some definitions for the terminology that will be used. Some of these definitions mirror those of the corresponding concepts introduced in Section 4.1.1 and are repeated as a reminder.

- *Entity*: A user, component, or group of components of the system that is subject to control. In the case of users, they are subject to control though their interactions with the system (events).

- *Attribute*: Any property of an entity that can be measured as a numeric value within a given range.

- *Information space* (Σ): A finite, discretized, and normalized cartesian space, where each dimension corresponds to an attribute: $\Sigma \subset \mathbb{Z}^{+^a}$.

- *Keyspace*: Label given to a set of attributes that together define an information space.

- *Policy definition space* (Σ_D): Information space composed of the keyspace of attributes related to management goals and/or utility.

- *Policy application space* (Σ_A): Information space composed of the keyspace of attributes related to management actions on system entities.

- *Point/Event*: Used interchangeably to denote a fixed set of attribute values, specifying a coordinate in an information space.

- *Region*: A region is a set $R \subseteq S$, such that any two points in R can be connected by a sequence of contiguous points in R. Each node is responsible for a unique region, denoted by node.region.

- *Cluster*: Let D be the data set being analyzed, then a set of points C is a cluster with respect to a region R, if density$(R, C) >$ density(S, D).

- *Centroid of a set* (centroid(X)): Point with minimum average distance to each of the points in X.

- *Width of a set along a dimension* (width(X, d)): The maximum distance of any point in X to centroid(X) along dimension d.

- *Profile element*: Either a point or cluster associated with a particular entity.

- *Profile*: A set of profile elements corresponding to the same entity.
- *Action*: A piece of code parameterized by an entity or an event.
- *Meta-policy*: A triple $< PE, R_D, A >$, where PE is a profile element, R_D is a region in Σ_D and A is an action.
- *Policy*: A triple $< E, R_A, A >$, where E is an entity, R_A is a region in Σ_A and A is an action.

We restate an overview of our approach in terms of the preceding definitions as follows. Managers define meta-policies associating actions to regions in Σ_D by way of a characterization of system state given by a profile element. The purpose of our framework is to a) enable this meta-policy definition; b) detect profile elements in monitoring data to be used with meta-policies; c) generate policies from meta-policies by mapping regions in Σ_D to regions in Σ_A using the properties of profile elements; and d) to enforce policies by applying the corresponding actions to events and entities that match policy regions.

5.1.2 Policy spaces

In most systems, policies are usually statically defined, with fixed constraint thresholds defined by managers, and pertaining to specific entities and system states. This is so because policies are defined in terms of high-level goals, as measured by attributes that act as indicators of these goals. Given these indicators, a system administrator can define a number of desirable or undesirable system states and determine a management action for each of these states with respect to the system goals. We consider applying management actions defined by policies to events composed of multiple attributes. A policy can then be defined by a region of application, an entity or entities to which it applies (i.e. the source of the event), and a management action. Figure 5.2 shows this concept.

The difficulty arises when defining the application region of the policy, because the relation between the event attributes that are being managed and the system states

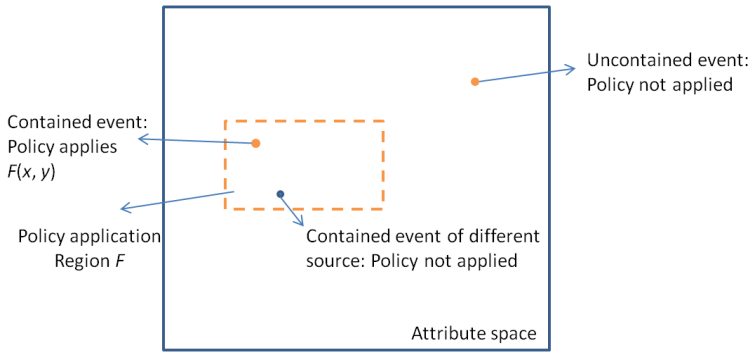

Figure 5.2: Representation of a policy as a region in a multidimensional information space

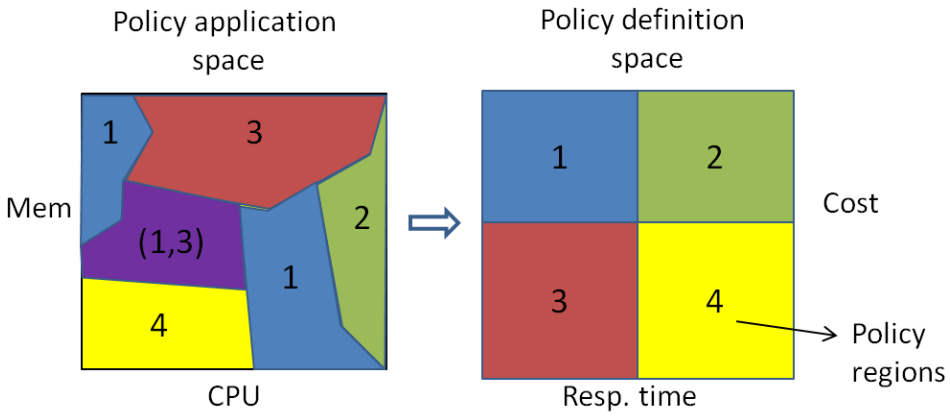

Figure 5.3: Two policy spaces with example attributes. Note that it is easier to delimit regions manually in the definition space (Σ_D) because of the nature of the attributes. The combined region (1,3) shows that one region in Σ_A may map to more than one region in Σ_D.

to which management actions are attached is not known and/or is dynamic. To see this more clearly, we can consider two distinct information spaces: one composed of attributes that are known indicators of system goals and where policy boundaries can be clearly delineated by an administrator, and another composed of managed attributes whose values are the input of management actions. The point made earlier is that these two spaces (i.e. the attributes that constitute them) are often not the same and are related in a dynamic way, dependent on the design of the system, workload and client interactions, available resources, etc. A snapshot of this dynamic relation is illustrated in Figure 5.3.

We call the first space the policy definition space and denote it by Σ_D, and the second

the policy application space, denoted by Σ_A. This is because, while an administrator can define a policy in terms of the attributes of Σ_D, these policies need to apply to events composed of attributes of Σ_A. Moreover, it is often the case that the values of events in Σ_D are not available a-priori, when the events in Σ_A take place, and are only available as feedback once an operation or system interaction has taken place (and after management actions should be applied).

In order to differentiate between policies in the two spaces, we refer to policies in Σ_D as meta-policies, because a meta-policy is actually a specification of when and how a dynamically generated policy should be applied. In order to derive policies from meta-policies one must approximate the mapping of attribute values between the two policy spaces. Our approach, illustrated in Figure 5.1, is to correlate both spaces by observing the feedback attributes for a sequence of monitored events and finding temporary mappings by finding agglomerations of patterns in these sequences that determine the dynamic boundaries of newly generated clusters. These agglomerations are found by applying clustering analysis to system events, composed of attributes from both spaces, and agglomerating the clustering results to generate policies in Σ_A from meta-policies in Σ_D. This approach is explained in more detail in the next section.

5.1.3 Clustering-based policy generation

Figure 5.3 showed an instance in time of the mapping between the policy application space (Σ_A) and the policy definition space (Σ_D). One way to partially obtain that mapping, as Figure 5.4 shows, is to find sequences of events that cluster in both spaces, because that would show a consistent relation between the regions covered by the clusters in each space. Such a clustering approach can only partially show the complete mapping at any point in time, since it is dependent on the events that occur within a given time window. However, it is useful because it conveys the mapping for events that, by definition of a cluster, are the most frequent or similar at that time, and agglomerations of such clusters over time will cover the regions that are most likely to contain events.

Consequently, it is necessary to find a cluster in Σ_A that is also a cluster in Σ_D (or

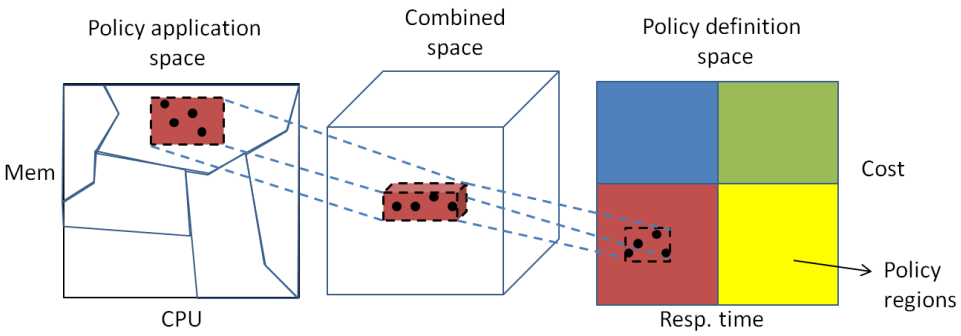

Figure 5.4: Clustering as a tool to discover parts of the mapping between policy spaces

Object	Properties	Methods
`Event`	`Source`	
`Region`	`Keyspace` `Ranges`	`Project(keyspace): Region` `Apply(Action)`

`Cluster`	`Size`
	`Density`
	`Centroid`
		`GetEntities()`
		`GetElements(Entity)`

`Profile`	`Entity` `Elements`	`Update(Element)`

Figure 5.5: Objects for meta-policy definition

viceversa). Sequences of events that are only clusters in one of the spaces are not useful because they do not show a clear trend that can be interpreted as part of the mapping. Clustering events in both spaces independently is, therefore, impractical. Instead, as Figure 5.4 shows, we perform clustering in a combined space, which is composed of attributes of both individual subspaces. Because clusters in this combined space will also be clusters in each of the subspaces, the meta-policy whose region of application contains the cluster in Σ_D can be applied to a region in Σ_A corresponding to the same cluster.

Figure 5.5 shows properties and procedure calls of the different objects that are used in a meta-policy definition. Please refer to Section 5.1.1 for detailed descriptions of the different objects. It is important to note from the operations that a region may be projected onto either space by removing some of its attributes. Therefore, a cluster that is found by the clustering algorithm has a region that may be projected onto both spaces.

For each managed entity in the system, we maintain a *profile* composed of *profile elements*. These profile elements are obtained by identifying the entities that produced the different events contained in a cluster (or a single outlier) and extracting those events for each entity (corresponding to the two cluster operations). It is important to consider outliers in meta-policy definitions because they indicate potentially anomalous and noteworthy behavior. Finally, meta-policies have specific *actions* that are application-dependent pieces of code that can operate on any of the objects and procedures listed in Figure 5.5.

Given all of the above objects and procedures, the cycle that includes the generation of a policy from a meta-policy is as follows (see Figure 5.6). Events are produced by managed entities to trigger actions in the system and are collected during a given time window. Policies, if they have already been defined, are applied to these events to influence the way that these actions are performed. Feedback from these actions, in the form of events in Σ_D is also collected during this time period. As the events are being collected, clustering analysis is performed using all available attributes. These results are available at the end of the time window, and are first projected onto Σ_D as profile elements. These profile elements trigger particular meta-policies by intersecting with R_D and contain the entities and other parameters (cluster size, location, etc.) that are parameters of the action in this policy. Finally, each final policy is generated by obtaining R_A as a projection onto Σ_A, and from the entities and actions in the meta-policy.

One issue that arises when generating policies and matching events to policies in this way is what to do when two policies are mapped to intersecting regions in Σ_A and additionally correspond to the same entity. In other words, there are potentially conflicting policies that can apply to the same set of events. Policy conflict resolution is outside the scope of our work, although it is possible to detect and/or resolve policy conflicts using manual techniques [51] or learning techniques [25], among others. However, it is important to note that additional conflicts may arise from inaccuracies in clustering and/or incomplete data. In Section 5.2, we use very simple conflict resolution techniques, but better techniques must be developed in future work.

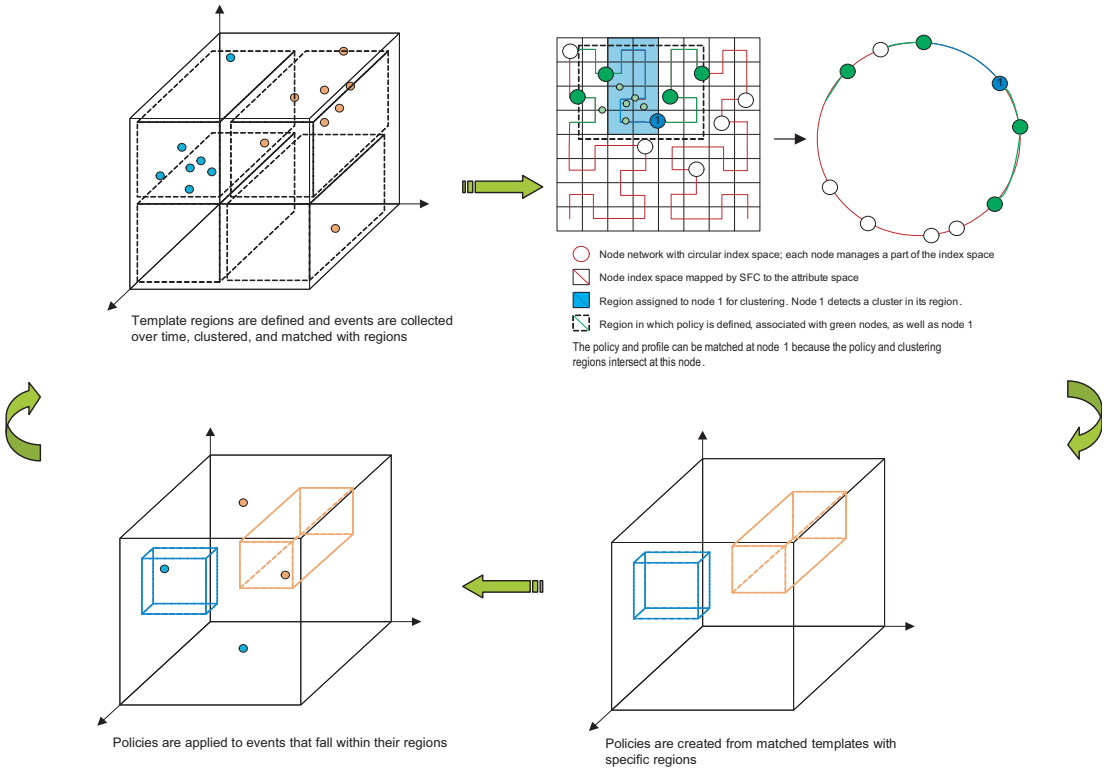

Figure 5.6: Application procedure for dynamic policies

5.2 Applications

The following application scenarios combine the use of different aspects of the decentralized online clustering and dynamic policy framework and illustrate the kind of autonomic management tasks that it can support.

5.2.1 Device Network Management

Monitoring and management of large fleets of multi-function devices (MFDs) that can provide physical services (printing, scanning, and communication and software services, such as fax, e-mail, and file storage and transformation) is important for characterizing usage patterns and load distribution, and answering questions about, for example, the relative bias of users towards devices due to their location, performance, and the services offered. Such understanding would help track the relative importance of devices and services, delineate possible over/under or malicious utilization of the network, and manage user QoS. Eventually, knowledge about user-device interactions in the network

derived through data analysis can enable on-the-fly optimization of task scheduling and service provisioning, and the isolation of malicious or anomalous users and system nodes.

These MFDs are equipped with enough computing capacity to harness for in-network management. To take advantage of this and demonstrate the use of the clustering and dynamic policy framework, we identify outliers in job submission data and apply policies for job priority assignment based on user/device job submission patterns.

The following experiments correspond to the analysis of real data collected from the operation of a device network consisting of 54 devices. The network actually serves an office environment that produces several hundred thousand pages a month. The data consists of descriptions of jobs submitted to the devices during a seven week period. In all, there were over 10,000 jobs of varying sizes (if the jobs were represented in terms of pages there would be several hundred thousand) in the dataset. During the seven week period, there were about 200 users interacting with these devices, which could be spread out over a small geographical area within a large organization spanning several buildings. The job descriptions identify the user submitting the job, the device receiving the job, and the job size, and include the date and time of the submission. The experiments correspond to two usage scenarios, described below.

Scenario 1: Monitoring device usage levels

In this scenario, the devices periodically monitor and report their usage level in order to compare and detect normal usage patterns as well as abnormally utilized devices. To this end, devices aggregate the job submissions for the given time period and produce a status update consisting of three dimensions: average job size, number of jobs received, and number of different users that submitted jobs to the device. Each one of these attributes is a different indicator of usage that can be interpreted and controlled in different ways.

In order to define the information space for this scenario, we must first define the unit distances d_i for each attribute. Recall that the unit distance is the minimum distance at which two attribute values are considered to be different, and that it is

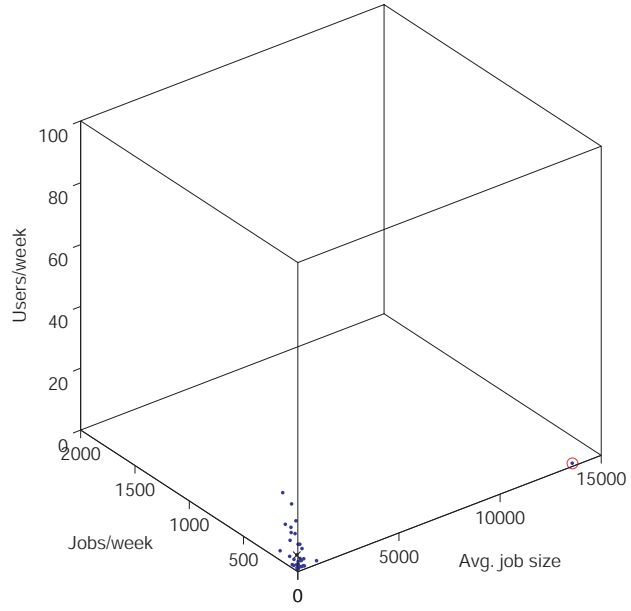

Figure 5.7: Initial analysis for one week period using uniformly distributed regions for the set of processing nodes. The figure illustrates the size of the information space compared to the region where points cluster, which cause processing node regions to be large initially.

strictly application dependent. In this case, we select $d_{jobSize}$ equal to 10KB because there were relatively fewer jobs below this value in our dataset. Since we wanted to analyze the patterns relating to individual users and jobs that were served/received by each device, $d_{numJobs}$ and $d_{numUsers}$ are both set to 1 - the smallest quantum for those dimensions.

The network is initially set up with 54 processing nodes (one for each device). One set of experiments was conducted setting the analysis period to one day, i.e. clustering was done on the data generated for each day, while for another the analysis was done for each week. In our dataset, the granularity of the weekly analysis provided better clarity, and is thus used for illustration. Figure 5.7 shows the results for a single week, in which events cluster near the origin, with a single outlier for a very large average job size. However, due to the large size of the information space, the processing node regions obtained with the initial parameter setting were large enough that nearly all data points ended up in the region corresponding to a single node.

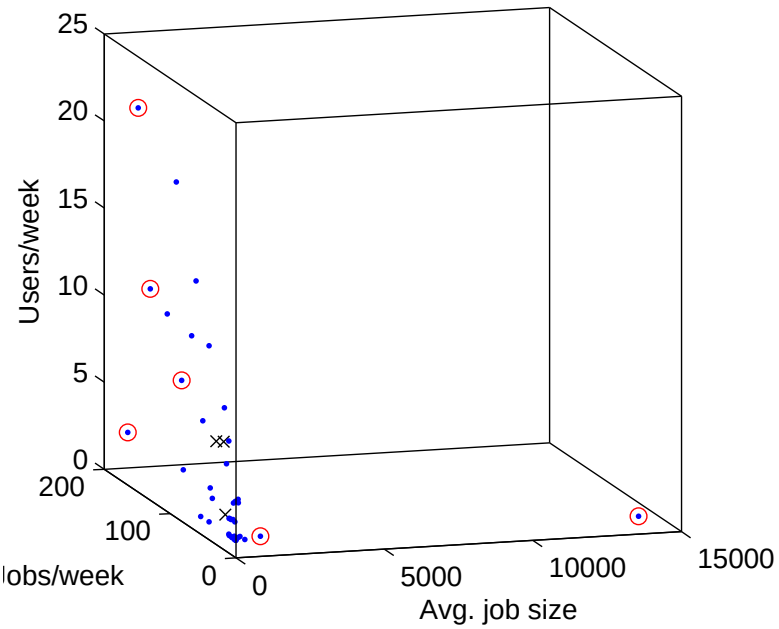

Figure 5.8: Analysis of one week period after increasing the number of regions near the origin

Given the initial clustering results, new region sizes can be calculated to obtain better cluster resolution near the origin. Figure 5.8 shows the new results of the clustering analysis for the same week as above. The scale has been modified to better visualize the new outliers that were identified. Figure 5.9 summarizes the results for the seven weeks of the experiment, showing only cluster centroids and outliers.

Additionally, outliers can be observed over several time periods to detect devices that have consistently abnormal usage with respect to the usage of other devices. During the seven week period, three devices were consistently present within the outlier reports for the system. Their status updates are highlighted in Figure 5.9. It can be inferred that the high usage of these devices is due to their importance within the network and/or preference of users toward them (e.g. because of high quality of job execution, strategic location in the system, kinds of services provided, etc.). This information can subsequently be used to allocate greater resources for those particular devices, or to improve other devices given their configuration/characteristics.

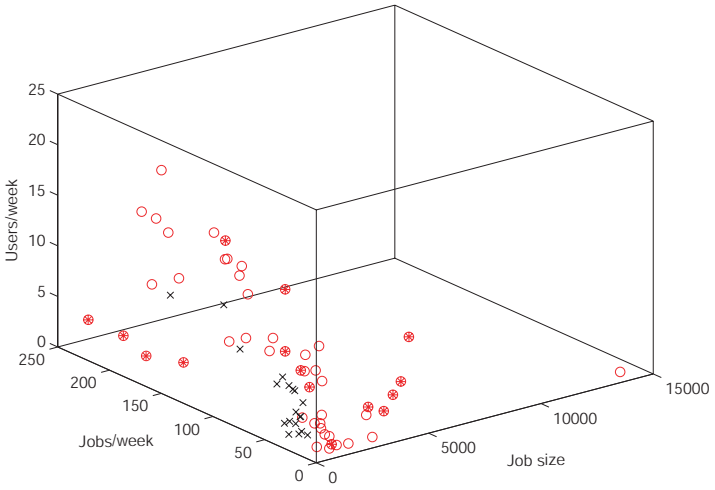

Figure 5.9: Summary of results for weekly periods (device usage). The filled circles represent the outliers from consistently anomalous devices.

Scenario 2: Usage control by monitoring job patterns

Usage Control defines policies that extend those of traditional access control by defining obligations and conditions, as well as authorizations, for users and policy application [62]. These obligations and conditions are intended to provide a richer set of semantics for runtime execution of the access control model by taking user actions and system context into account for the application of authorizations.

In this type of device network, it is hard to come up with consistent resource quotas and access privileges a priori because of the following: a) Most users (e.g. people who are printing, faxing, scanning and using other document services) have their own unique time-of-day dependent usage profile which deviates significantly from other users; b) There is a distance element to most document services on the device network as people have to walk up to their device and hence it is not easy to predict the user-device mapping; and c) Since these devices are internal to the organization, most users of services have more than an adequate (in some other cases a very restricted) set of privileges assigned to them, causing cost inefficiencies and security threats. As a result, administering polices and accounting for changing user behavior is difficult in this domain.

We can show the dynamism of policy generation using our approach by defining

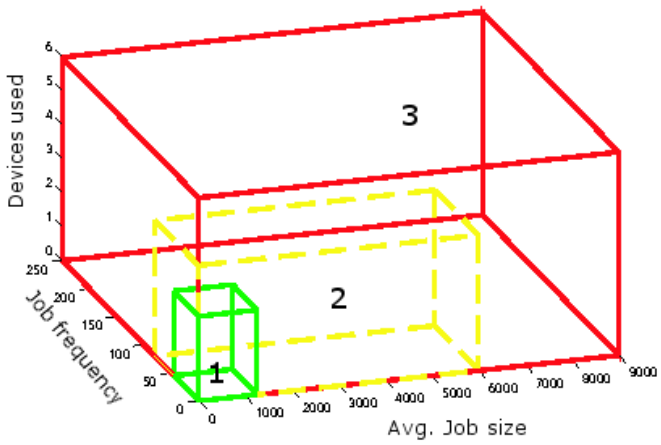

Figure 5.10: Representations of the meta-policy regions as a priority mapping within the definition space

meta-policies for this scenario. In order to do this, it is necessary to first identify the attributes for Σ_D, according to the management goals. In this case, the system goal is having a fair and balanced use of resources in the enterprise. In terms of usage control, the rights to use certain device services (objects) by particular users (subjects) are dependent on a user's agreement to utilize the services in consideration of the needs of other users and on the load conditions of the system as a whole.

The attributes that describe the usage trends of the system over time (obtainable from job requests) are the average job size of a user within a given time period, the number of distinct devices printed to by the user, and the job submission frequency of the user in the period of time. A system administrator, based on experience, can set regions of interest in the definition space that indicate the perceived fairness of situations in which clusters of particular characteristics fall within these regions. Possible regions for this scenario are shown in Figure 5.10.

A simple meta-policy that can be defined based on these attributes is one that assigns priorities to jobs according to usage profiles and device utilization. For example, jobs from low impact users (infrequent small jobs) to underloaded devices can be given a greater priority than jobs from higher impact users and/or to more highly loaded devices. This meta-policy can be realized by associating increasing priorities from the origin to the regions of Figure 5.10, and, though simple, is suitable for illustrating the

dynamism in policy generation.

The application space in this scenario is composed of request attributes of individual job requests. Namely, the attributes are job size and the device to which the request is sent. One thing to note is that applying clustering analysis to device IDs would not yield very interesting results unless these IDs we related in some way. In this case, device IDs are grouped based on the proximity of the devices, so that clusters along this dimension indicate jobs submitted to devices that are close to each other. This would be useful, for example, for defining policies that allow the redirection of jobs to nearby devices.

For this experiment, we divided the data set into three chunks. The first two correspond exclusively to jobs that originate in the morning and in the evening, respectively, and are used as input to the clustering algorithm to obtain the policies, which emerge as agglomerations of regions in Σ_A (shown in Figures 5.11 and 5.12). The colors and numbers indicate the mapping of the meta-policy regions and indicate the priorities given a-priori to jobs of particular characteristics. Devices nearer the origin are those that are closest together, and thus exhibit greater clustering of jobs. Note that these devices are, in general, less loaded in the morning than in the evening, shown by the large regions of high priority in the former, compared to a lower priority in the evening. Also note that in the morning, regions of lowest priority are found for a group of devices, both for large and small jobs. This can be explained by the behavior of certain users submitting jobs of these characteristics at a high rate to these devices.

The third chunk of data is used as a test set to which the two different generated policies are applied. Note that both policies were obtained using the same meta-policy and merely reflect the dynamic mapping due to time-of-day usage differences. The result of this experiment is shown in Figure 5.13 as two separate priority distributions of the submitted jobs. This means that, for the same sequence of jobs, individual jobs are be assigned different priorities based only on their application space attributes and the automatically generated policies.

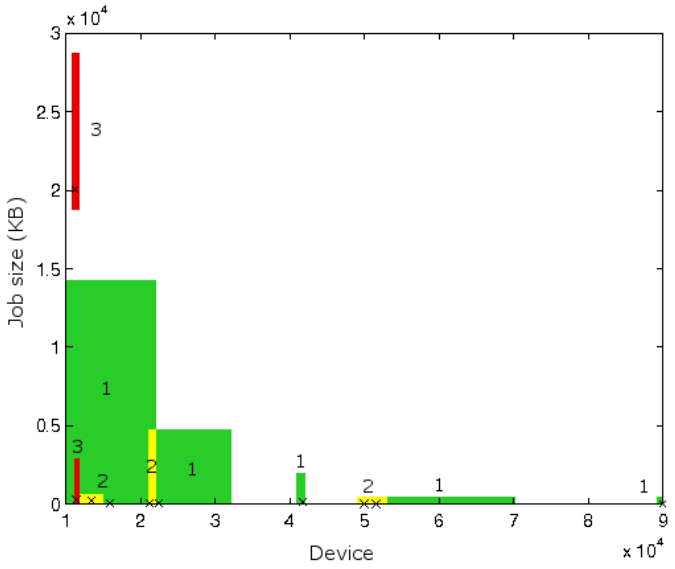

Figure 5.11: Policy regions generated based on clustering the events originating in the morning, where the priority of each region is indicated by the numbers (1-3) or the colors of the region boxes.

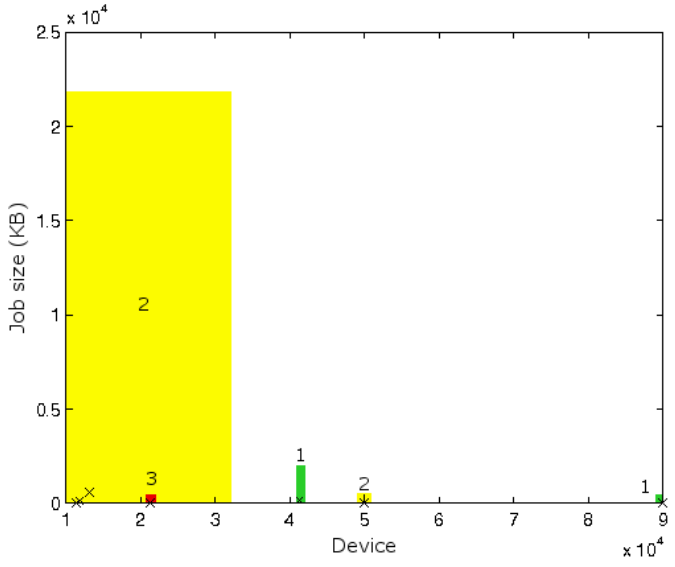

Figure 5.12: Policy regions generated based on clustering the events originating in the evening, where the priority of each region is indicated by the numbers (1-3) or the colors of the region boxes.

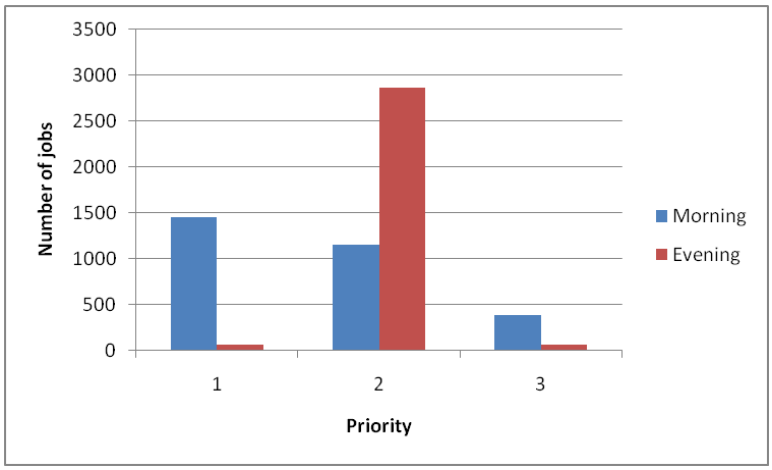

Figure 5.13: Priority distributions obtained by applying morning and evening policies separately to a single job sequence

5.2.2 VM Provisioning

The emergence of cloud computing is leading to a paradigm shift in the way that research centers, enterprises and businesses are viewing their IT and computational needs, i.e., many organizations are considering using clouds to support a significant portion of these needs. This in turn is resulting in relatively long term, but highly varied and elastic (dynamically growing and shrinking) demands for cloud computational resources from various organizations of different types and sizes. At the cloud providers' end, consolidated and virtualized data centers attempt to exploit economies of scale and provide virtual machine (VM) containers to host a wide range of applications and provide users with an abstraction of unlimited computing resources - users can essentially "rent" VMs with required configurations to service their requests.

As scales, operating costs, and energy requirements of these data centers increase, maximizing efficiency, cost-effectiveness, and utilization becomes paramount. However, balancing QoS guarantees with efficiency and utilization becomes extremely challenging, especially when the workloads are highly dynamic and extremely varied. A typical workload in such a cloud infrastructure will consist of a dynamic mix of heterogeneous applications, such as long running computationally intensive jobs, bursty and response-time sensitive WS requests, and data and IO-intensive analytics tasks. Therefore,

even though ensuring QoS and responsiveness can typically be accomplished using pre-allocated VMs and overprovisioning, the increasing difference between these resource allocations and application requirements magnifies resource under-utilization and leads to cost increases. Consequently, there is a growing need for providing more reactive on-demand provisioning, leading to better efficiency and utilization, but without an unacceptable impact on system responsiveness.

In a cloud environment, executing application requests on underlying Grid resources consists of two key steps. The first, which we call VM Provisioning, consists of creating VM instances to host each application request, matching the specific characteristics and requirements of the request. The second step is mapping and scheduling these requests onto distributed physical resources (Resource Provisioning). While much work to date has been dedicated to the resource provisioning problem, which is not addressed here, under-utilization of resources resulting from inappropriate VM provisioning is still a relatively unexplored problem. Most virtualized data centers currently provide a set of general-purpose VM classes with generic resource configurations, which quickly become insufficient to support the highly varied and interleaved workloads described above. For example, Amazon's EC2 only supports five basic VM types [1]. Furthermore, clients (enterprises and SMBs) can easily under or overestimate their needs because of a lack of understanding of application requirements due to application complexity and/or uncertainty.

In this scenario, DOC can be used to efficiently characterize dynamic (rather than generic) classes of resource requirements and can be used for proactive VM provisioning, with the goal of reducing overprovisioning caused by the difference between the virtual resources allocated to VM instances and those requested by individual jobs. Furthermore, policies can be defined to address the inaccuracies in client resource requests that lead to inaccurate provisioning.

Clustering-based VM Provisioning

VM provisioning refers to the efficient allocation of virtual resources to application jobs as they arrive data center service queues, through the creation and allocation of appropriately configured VM instances. Enterprise Grids hosting cloud services typically have multiple geographically distributed service providers and entry points, i.e., multiple queues may exist, with users submitting requests to their local application queues. The state of a system in terms of resource requirements is represented in the short-term by the job requests from different applications, as they wait in the multiple queues. A dynamic provisioning solution needs to create VM instances by closely approximating the specific resource requirements contained in this state. However, a reactive approach in which each individual request results in a custom VM allocation would be highly inefficient because of the significant wait times incurred in the instantiation of new VMs, in addition to the complexity of dynamically allocating physical resources for such a wide range of resource configurations. According to a recent study [60], the instantiation and startup times of VMs on Amazon's EC2 are in the order of 60–120 seconds. Thus, techniques such as DOC that can provide an online analysis and accurate characterization of the dynamic state within this time frame can provide a set of required resource configurations that can be proactively provisioned and allocated.

The proposed mechanism for dynamic and decentralized VM provisioning is as follows. As with most predictive approaches, we divide the flow of arriving jobs into periods (analysis windows). In each window, an instance of the DOC algorithm is run on the jobs that arrive, producing a number of clusters or VM classes. At the same time, each job is assigned to an available VM as it arrives, if one has been provisioned with sufficient resources to meet its requirements. The provisioning is done based on the most recent analysis results from the previous analysis window, as follows.

For the first window, the best option is to reactively create VMs for incoming jobs. However, the job descriptions are sent to the processing node network and by the end of the analysis window each node can quickly determine if a cluster exists in its particular region. If so, the node can locally trigger the creation of new VMs for the

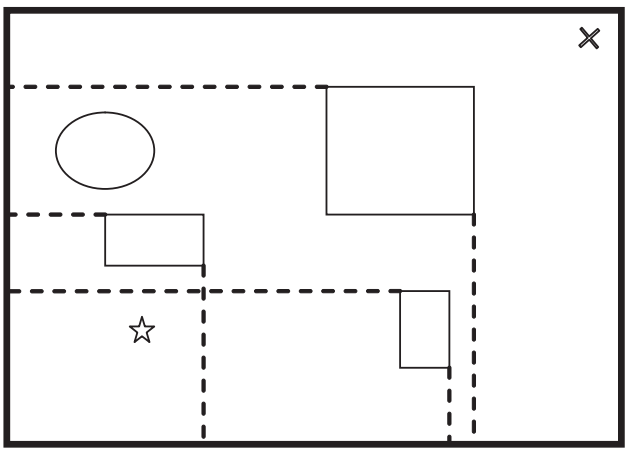

Figure 5.14: Example information space and clustering results for dynamic VM provisioning.

jobs in the next analysis window with similar resource requirements. According to [60], the time required to create batches of VMs in a cloud infrastructure does not differ significantly from the time for creating single VM instances. Thus, the VMs for each class can be provisioned within the given time window. Because the classes correspond to a characterization of incoming requests in the short term, it is expected that not all windows will require reprovisioning.

To clarify the above procedure, consider an example illustrated by Figure 5.14. The figure shows a two-dimensional information space with three initial clusters (squares). The top right-hand corner of these squares represents the resource values that characterize the class of jobs belonging to that cluster (because this amount of resources is sufficient for all of these jobs). The dotted lines represent the regions defined by each cluster that are mapped to the nodes that will contain the corresponding VM descriptors. Any job that falls within those regions (like the star in the figure) can be automatically assigned to the closest VM type (overlapping regions have more than one VM type description). The circle is a cluster that is discovered in a subsequent time window. This cluster can be used to create and provision a new VM type. The **x** is a request that falls outside existing regions (outlier) and represents a job whose requirements exceed the currently provisioned resources. In this case, the wait cost of reactively creating a new VM type would have to be incurred.

To test this approach, we examined the Los Alamos National Lab (LANL) CM-5 log, obtained from [28], which contains 201,387 jobs during two years. We used this workload because computationally intensive applications are increasingly becoming part of enterprise datacenter and cloud workloads. The requests from the LANL workload contain the jobs' resource requirements (memory, number of CPUs (N), and requested CPU time (T)). For clustering, we use memory and derive a value from the last two requirements that represents CPU speed (C). The value is calculated as:

$$C = \frac{N \times T}{100}$$

The value of 100 in the above calculation is a normalization factor that represents the duration in seconds of a generic or reference job. Multiplying the number of CPUs and requested CPU time gives a total CPU time requirement. The calculated CPU speed corresponds to the speed necessary to finish each job in the fixed reference time. We divide the dataset into several sequential subsets of job requests (of 1000 jobs each in our case, chosen as an extreme value), each of which represents the contents of the queues to be analyzed for provisioning within each analysis window.

The first experiment, which measures only overprovisioning cost, is meant to show the accuracy with which the different data analysis approaches can statically approximate the resource requirements of a job set. The DOC approach is run against centralized k-means clustering, another commonly used algorithm (see Section 2.2.3) that is expected to provide a benchmark in terms of clustering accuracy. As a reference point, we also consider the overprovisioning approach. For the reactive approach, of course, no overprovisioning cost is ever incurred. The graphs in Figure 5.15 show the overprovisioning cost for resource R (CPU speed and memory) in each window i as the average difference between requested and provisioned resources for the set of jobs j in that window. This is:

$$O_i^R = \frac{1}{N} \sum_{j=1}^{N} \left(P_{ij}^R - Q_{ij}^R \right)$$

On average, both DOC and k-means approaches perform similarly, which is a good result, considering that decentralization and online execution must be traded for some

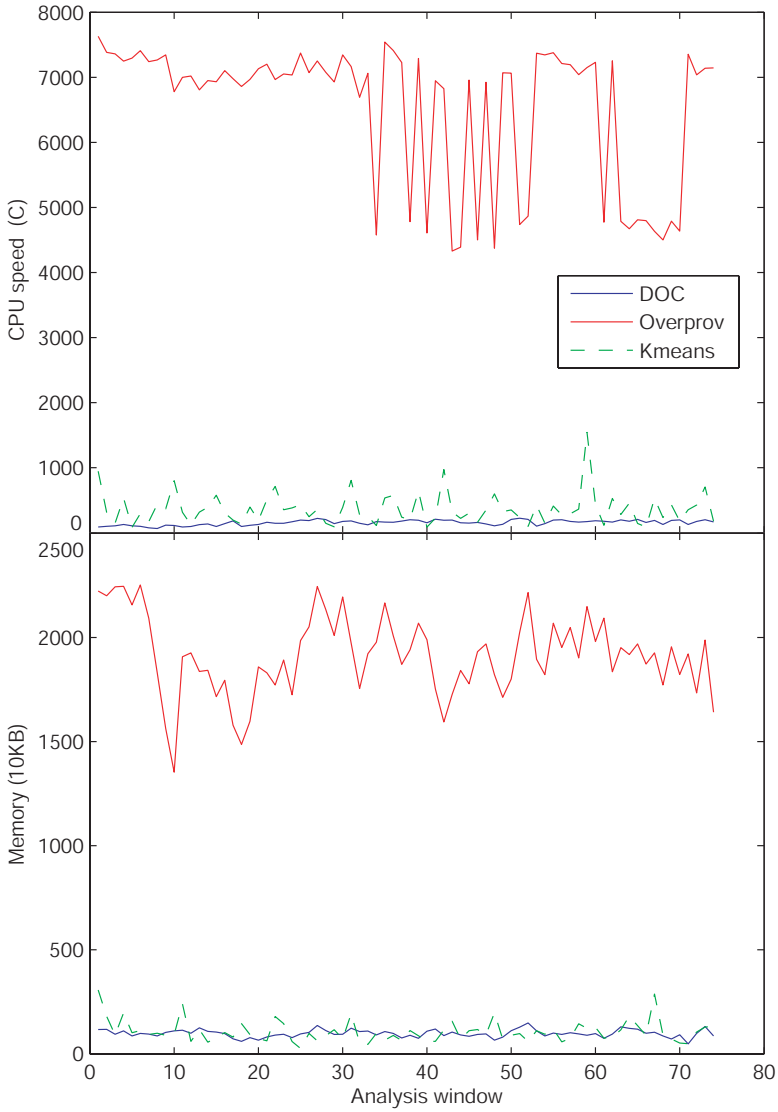

Figure 5.15: Average difference between requested resource (CPU speed (top) and memory (bottom)) for jobs and the corresponding analysis result in each time window

accuracy in the clustering results (see Section 4.2.4). In fact, there is more variation in the results of k-means, which we attribute to the differences in the selection of k. In this experiment, we ran the k-means algorithm for a large range of k values and picked the best result for each run. This is a disadvantage with respect to the density-based approach, since this procedure for determining k values is usually done offline.

The dynamic application of the different approaches with respect to provisioning response times is evaluated in the second experiment, in which the job sets form a sequence of incoming requests and the analysis results in each window are used for provisioning for the set of jobs in the next window. The impact on response time of each approach is measured by counting the number of VMs required by each job that could not be preallocated based on the predictions of that approach. This is because each such VM must be created after the job is received and not during the previous analysis window. Figure 5.16 compares the approaches that are amenable for online execution, which are the reactive, overprovisioning, and DOC approaches. In all of these, the VMs for the first set of jobs must be allocated reactively, so that all 1000 jobs count in the first analysis window. Afterward, however, the overprovisioning approach was always able to proactively allocate VMs for every job, so that its cost levels off at 1000. In contrast, all VMs created by the reactive approach are counted. The DOC approach performs very closely to the overprovisioning approach, incurring creation cost for outlying jobs or whenever the number of VMs provisioned for a resource class is not sufficient. The total count for the DOC approach is 1462 out of 45000 in the time frame shown.

Energy-Aware Provisioning

The flexibility afforded by clustering-based VM provisioning for determining resource configuration classes can be exploited from an energy perspective by using these resource classes to influence the selection or configuration of physical resources in a resource provisioning strategy. The optimization of energy efficiency results from two distinct strategies. The first is a static strategy that consists of closely matching the resource usage of the VMs to the resource characteristics of the available physical servers. This

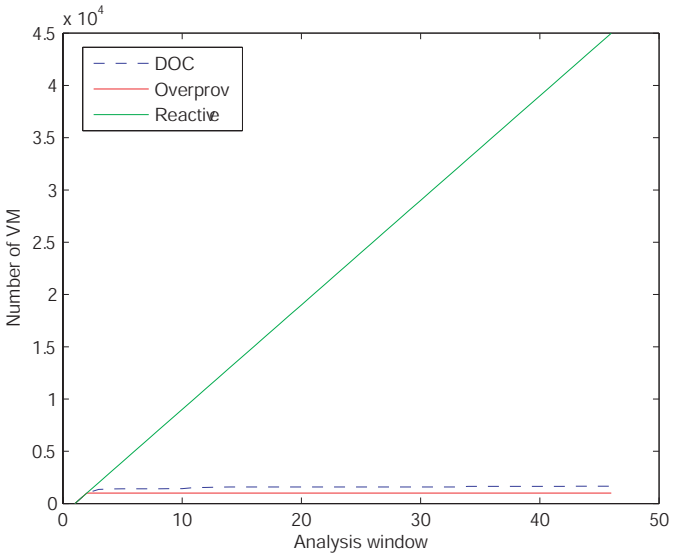

Figure 5.16: VM creation cost vs. time for the different approaches

is essentially accomplished by mapping all of the jobs from single or closely related VM classes (clusters) to similar machines. The second strategy is a dynamic approach that requires being able to scale or power down specific subsystems of the physical resources when they are not strictly necessary, given the mapping of specific VM classes to the physical servers.

The static an dynamic approaches can be combined, by assigning VM classes to servers with configurations that are close enough to those required, and only re-configuring and provisioning available physical servers when the configurations of other servers differ significantly. We only consider reconfiguring idle servers, except for upgrading servers that are configured with fewer subsystems or less resources than those requested.

The dynamic server configuration strategy depends on the existence of power control capabilities in specific subsystems, such as:

- *CPU*: The speed of the CPU can be controlled via Dynamic Voltage Scaling (DVS). For applications that are memory bounded, the control of CPU speed using DVS can lead to an affordable reduction in energy consumption.

- *Memory*: The energy consumption of the memory modules can be reduced by controlling the memory frequency. For those applications that do not require

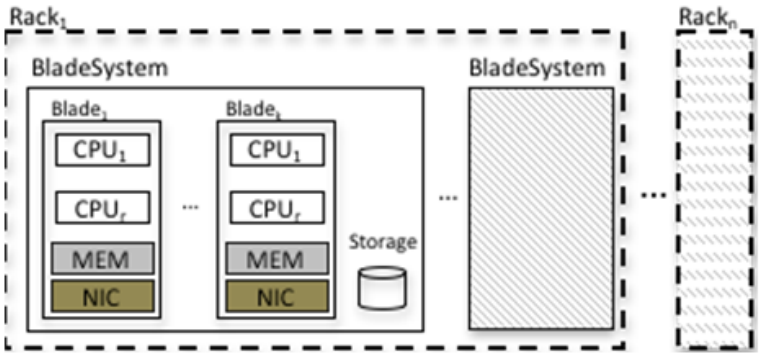

Figure 5.17: Blade data center architecture model

high memory bandwidth, it is affordable to reduce their frequency or eventually shut down some banks of memory in order to save energy.

- *High performance storage*: Disks can be powered down and possibly replaced with flash memory devices that require less power when applications are not I/O intensive.

- *High performance network interfaces*: Certain NIC subsystems (for example Myrinet interfaces) can be powered down for non-network bound applications.

Current datacenters usually rely on modular architectures, such as those based on blade servers, that bring new efficiencies and flexibility. Current datacenters commonly use racks composed of blade server chassis (such as IBM BladeCenter or HP BladeSystem) containing different blade server nodes. Compared to traditional 1U rack servers, blade servers provide flexible ecosystems, easy maintenance and on-demand scalability. Moreover, this configuration allows a higher number of processors and memory while requiring less energy [83, 84]. Therefore, as is shown in Figure 5.17, it allows the configuration of specific subsystems per blade system (e.g. storage) or per node individually (CPUs, memory and NIC).

The application of clustering-based VM provisioning to energy-aware resource provisioning is the subject of current work, the initial results of which can be found in [73]. These evaluations were based on simulations and using workload traces from highly

distributed production systems. Compared to typical reactive or pre-defined provisioning, our approach achieves improvements in energy efficiency of 15% on average, with an acceptable penalty in QoS (less than 5% in workload execution time). However, we recognized limitations of the approach due to the sporadic unavailability of configurable servers for new VM classes. Therefore, additional mechanisms such as dynamic VM management (such as VM migration) are necessary to improve our approach.

Request Management Policies

The results of the analysis of resource requirements for VM provisioning can only be as good as the estimations of these requirements provided by job requests, as they relate to performance and response times. These estimations are either based on experience by submitting users or are approximated using historical data. In this scenario, the goal is to apply policies to prevent over or underestimation of required resources by making sure that the incoming job requests are as accurate as possible. (as close as possible to the actual resource consumption of the job). We will show that this goal can be effectively achieved using our dynamic policy approach.

Meta-policies in this scenario can be defined with respect to the slack in the requested vs consumed resources of the jobs submitted to the data center. To achieve the goal of minimizing slack, the policies that will be generated will actually modify the requested values contained in incoming jobs, before those jobs are serviced. The slack attributes that constitute Σ_D are obtained by dividing the requested and actual resource values. One thing we noticed from the data is that the number of CPUs requested is almost invariably granted, and there is no way for an application to use more CPUs than it has been given to run on. Therefore, we do not use this attribute and instead focus only on computation time and memory.

Figure 5.18 shows the meta-policy regions in Σ_D. Basically, each region has ranges representing different magnitudes of slack for computation time and memory, so that different transformations can be made to incoming requests accordingly. For example, an aggressive meta-policy (large transformation) can be defined beyond a certain slack

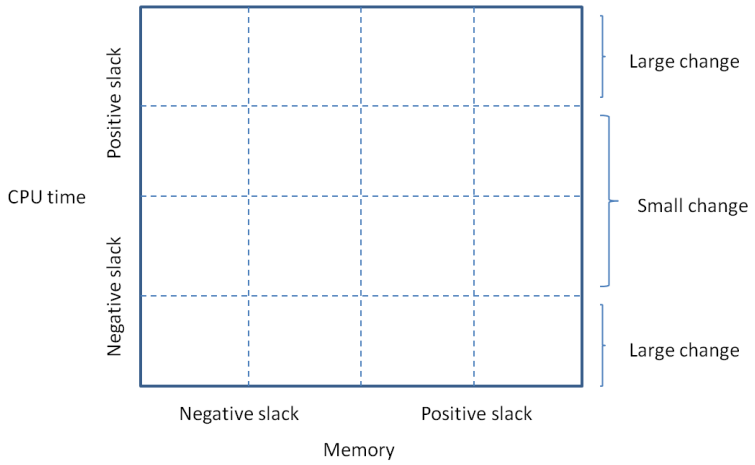

Figure 5.18: Definition space with meta-policy regions for the resource management scenario

threshold for both resources. Conversely, when the slack is small, a less aggressive meta-policy can be defined to avoid overshooting the difference. The exact transformation values of the generated policies depend on the profiles elements (in this case, clusters) that fall into each region. For the present evaluation, we used the cluster width as a parameter for the transformation: aggressive transformations used the upper boundaries of the cluster regions (by adding the width to the cluster centroid), while less aggressive ones used the lower boundaries.

To generate the management policies from these meta-policies, we divided the job trace into chunks as in the previous scenario, representing the clustering windows, and then playing back the events to apply the generated policies. Note that it does not matter that we used the same trace for analysis and simulation, since, in fact, this is an online approach, and the analysis of any given window is only applied to events of subsequent windows. We also assume that the application of policies does not influence future events (which could happen if users modified their requests based on observing the transformations applied to past events). This assumption does not compromise our evaluation, since, in fact, it furthers the case that a solution that can handle such dynamics is needed.

Figures 5.19 and 5.20 show the reduction in slack for compute time and memory, respectively, with respect to the request/consumption difference of the actual jobs

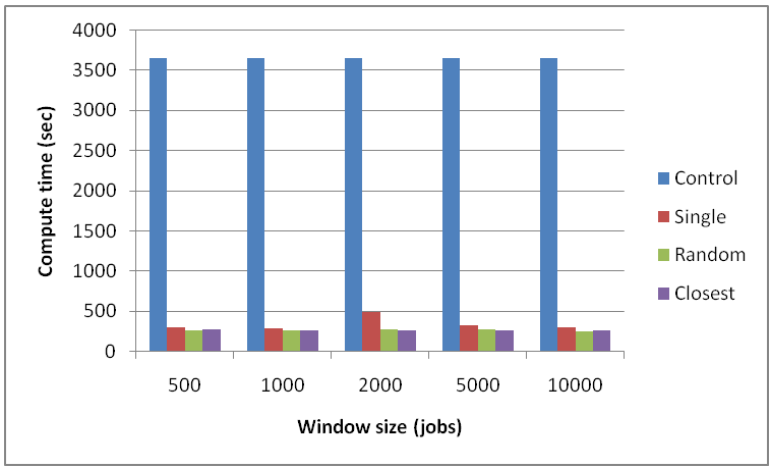

Figure 5.19: Average computation time slack for the jobs in the trace

(control). The reduction in compute time is clearly significant, and, while there is a reduction in memory, the slack for this resource was not very large to begin with. There is also an improvement when analysis windows get larger, but the improvement is either insignificant (for compute time) or actually starts to revert after a certain window size (for memory). This supports the online clustering approach, as opposed to static/offline analysis of traces, since trends change dynamically and old policies must be replaced by new ones in oder to be accurate.

Note that three different conflict resolution strategies were used for when two or more different policies applied to a single event. As a hypothetic comparison, we used a strategy where the closest meta-policy region to the actual resource consumption value of the job is chosen, if this consumption were known. The other two strategies corresponded to applying policies only when there were no conflicts (single), or choosing a policy randomly from the ones available. In all cases, the single strategy was similar to the closest strategy. Curiously, the random strategy did better than the closest in some cases, possibly because the closest strategy may have been too aggressive, overshooting the slack difference when the values are close.

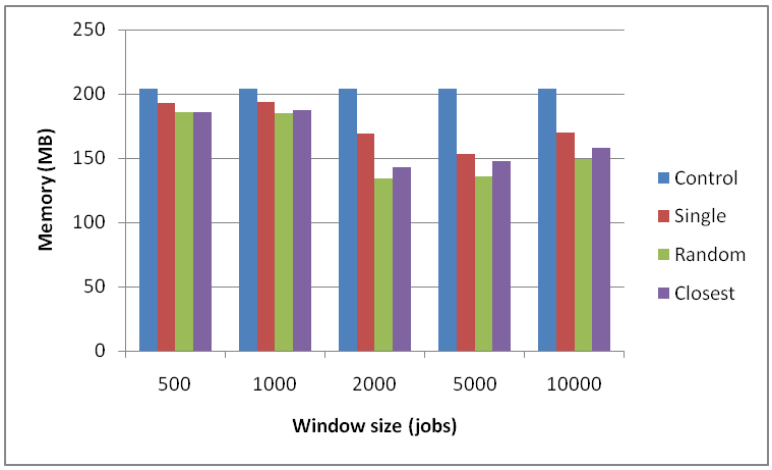

Figure 5.20: Average memory slack for the jobs in the trace

Chapter 6

Conclusions

The research presented in this document explored decentralized online clustering for the autonomic management of distributed systems and included work on three levels. The first is the decentralized, density-based clustering algorithm that spatially maps distributed data onto processing nodes and then aggregates local results, which include clusters as well as outliers (noise) in the data. The second is the implementation of this algorithm on top of a communication infrastructure based on a structured, content-addressed overlay for the robust and efficient distribution of clustering data and the aggregation of results. The third level is the application of decentralized online clustering to distributed system management, performing system monitoring, online data analysis, clustering-based profiling, and dynamic policy management. This work was applied to problems of anomaly detection, autonomic resource provisioning for large-scale data centers and clouds, and usage control for device networks.

The basic approach for cluster detection consists of evaluating the relative density of points within a multidimensional information space corresponding to a set of monitored attributes. In order to evaluate point density, the information space is divided into regions, and the number of points within each region is observed by an individual processing node. With knowledge of the total number of points in the information space, processing nodes recognize clusters and outliers in their regions by comparing the density of points contained therein with a baseline density for a uniform distribution of points. The calculation of the baseline density corresponds to a novel parameterization that does not rely on data-dependent information, facilitating the dynamic and automated application of the clustering algorithm. It was demonstrated that the clustering algorithm itself requires minimal computation at processing nodes, which makes it suitable for online execution. When compared to other known centralized clustering algorithms in terms of accuracy, the proposed decentralized online clustering (DOC)

algorithm never had more than a 15% deficit and even performed better on some data sets. In fact, it was shown that the data distribution had a greater impact on accuracy than the algorithms themselves. Furthermore, for a online monitoring scenario in which data is initially distributed, it was shown that the impact on accuracy is compensated for by the cost of centralization.

The performance of the decentralized clustering algorithm relies considerably on the content-based messaging substrate that supports the exchange of status updates and the partitioning of the information space into regions by implementing a dynamic mapping of points in the information space to processing nodes. The performance and data distribution evaluations relied on the capacity of the messaging substrate for getting the information used by the clustering analysis to the distributed processing nodes in a scalable fashion. Work on the communication framework also dealt with the robustness of the decentralized mechanisms at three levels. First, the overlay network that supports the messaging substrate guarantees the routing of messages and maintenance of the overlay structure despite membership changes and node failures. At the data messaging level, we used the concept of replication chains to replicate and recover data points to ensure their availability to the clustering algorithm. Finally, instead of having replication chains of a fixed length, we used the concept of selective replication to probabilistically replicate only those points that are likely to have an impact on the result of the clustering analysis, using Bayesian probabilities and historical data. Our evaluations showed that our heuristic replication model can bound the loss probability of important points while maintaining a short expected length of replication chains.

Finally, we also explored how to use clustering results for decision-making, as a way to close the loop of the autonomic management cycle. This is done via the dynamic application of management policies defined in terms of profiles constructed from online analysis. The mechanisms developed were applied to different use cases, demonstrating the value of our proposed approach.

For device network management, the clustering analysis of job submission data allowed the identification of consistent abnormal use of particular devices with respect to the rest of the network, indicating the criticality of these devices. Additionally, the

application of dynamic policies for assigning priorities to job submissions with respect to user behavior and device usage uncovered a dependency of job submission patterns with the time of day, which was accounted for in the policy generation.

For data center resource management, we explored the use of clustering for VM provisioning, energy-aware resource provisioning, and the application of request management policies. The provisioning strategies helped reduce the overprovisioning and energy cost in the allocation of VM resources by 1) allocating resources based on the classification of incoming resource requests, as opposed to the assignment to predefined VM configurations; and 2) scaling or powering down physical servers based on the resource requirements of the VM classes assigned to them. Furthermore, the accuracy of VM provisioning was improved (by an order of magnitude for compute time and up to 30% for memory) by modifying inaccurate incoming resource requests using dynamic policies.

We reiterate the specific contributions of this work:

- A decentralized online clustering algorithm and its robust implementation based on a distributed monitoring and messaging infrastructure.
- Evaluation of the clustering algorithm and the tradeoffs and conditions for its application.
- A conceptual framework for the creation of system profiles from clustering results and the definition and dynamic application of management policies.
- Demonstration of applicability and results from actual use cases and system data.
- Application framework for the deployment of the self-monitoring, clustering, and dynamic policy generation for autonomic distributed system management.

6.1 Milestones and Future Work

6.1.1 Load balancing and processing node management

By definition, data that is clustered is distributed in an unbalanced way in the information space. Because our clustering approach is based on the spatial distribution of data, nodes that are responsible for regions in which data is clustering will be more heavily loaded than others in terms of data and messaging. Although we have designed and implemented mechanisms for load balancing, which have been employed in our accuracy evaluation runs, we have yet to specifically evaluate both the extent and the impact of this load imbalance and the tradeoffs with the overhead of the load balancing solution.

6.1.2 Definition and Dynamic Application of Management Policies

As described in Section 5.1, we have explored ways to dynamically associate and adapt policies to system profiles obtained by clustering, so that the application of these policies is tailored to the dynamic behavior of managed elements. We have also developed ways to use our distributed platform to map policies to regions in the clustering information space, and consequently to profiles that fall within that region.

At present, our policy definition language is conceptual, and we rely on hand-coded class files to specify meta-policy definitions for use in our framework. The natural next step in this work is to develop a script-based language to write and interpret meta-policies. The cluster-based mapping of meta-policies to policies in the application space has been demonstrated empirically on specific use cases, but a more general and analytic study is needed in order to obtain bounds on the accuracy of this mapping. This is especially necessary to explore and develop mechanisms to address the generation of conflicting policies, arising from the use of incomplete, online data.

6.1.3 Prediction and Learning Mechanisms

Ultimately, the goal of an autonomic system is to be able to respond to unexpected situations, for which learning and prediction can play a crucial role. Online, profile based

prediction and dynamic policy application exploit the predictive potential of trends indicated by clusters. However, we believe that learning and model-based prediction can be more explicitly applied to several elements in our autonomic management framework, such as data distribution learning for improving the accuracy of clustering results, and probabilistic modeling, in which monitoring is viewed as a sampling problem due to system scale and trust/security issues.

Our clustering algorithm is based on the comparison between a uniform distribution of data in the information space and the actual possibly clustered distribution of system events. Because we use online monitoring to analyze events periodically over time, previous clustering results can be used to reshape the initially assumed distribution to improve clustering accuracy and outlier detection. In other words, the clustering algorithm can learn the data distribution from previous results. To this end, techniques for managing historical data must be explored.

Because profiles constitute characteristic system behavior, they can be used for history-based prediction. For example, if a user profile has information about resource utilization, which clusters over time, the cluster centroid can be used as a prediction of future resource utilization for effective resource provisioning. Attribute correlations that can be obtained from profile information can also be used to construct models from existing data, to predict, for example, missing attribute values from incomplete data, or to better detect outliers. Although this aspect has been exploited to some extent in the management scenarios with dynamic policies, further work using different modeling and prediction techniques is worthwhile.

Given large systems with a very large number of components, it may not be feasible to analyze all of the event data that originates from monitoring, even in a distributed setting. It is also possible that system components must be queried for status information, or monitored remotely because of trust issues. In this sense, allocating resources for monitoring and data collection can be seen as a sampling problem, for which different probabilistic techniques have been devised. Methods such as probabilistic collocation in the field of engineering design are of special interest because they can be used to model uncertainty in system response and because they are claimed to be faster with respect to

other related methods. These techniques should be explored with the purpose of using clustering results and the resulting data distributions as probability distributions that can guide the sampling of event information, considering their feasibility for in-network and decentralized application.

References

[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[2] Arnaldo J. Abrantes and Jorge S. Marques. A method for dynamic clustering of data. In *British Machine Vision Conference*, 1998.

[3] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*, 2003.

[4] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of 1998 ACM-SIGMOD Int. Conf. Management of Data*, pages 94–105, Seattle, Washington, June 1998.

[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] R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*, chapter Introduction to data structures and algorithms related to information retrieval, pages 13–27. Prentice-Hall, Inc., Upper Saddle River, NJ,, 1992.

[7] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, and Eugene Spafford. An architecture for intrusion detection using autonomous agents. Technical report, Dept. of Computer Science, Purdue University, 1998.

[8] 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.

[9] Steffan Baron and Myra Spiliopoulou. Temporal evolution and local patterns. *LNCS: Local Pattern Detection*, 3539:190–206, 2005.

[10] G.E. Box, W.G. Hunter, and J.S. Hunter. *Statistics for Experimenters: Design, Innovation, and Discovery*. Wiley, 2005.

[11] George Box and Gwilym Jenkins. *Time Series Analysis: Forecasting and Control*. Prentice Hall, 1994.

[12] J.M. Bradshaw. *Software Agents*. AAAI Press/The MIT Press, 1997.

[13] David Breitgand, Rami Cohen, Amir Nahir, and Danny Raz. Using the right amount of monitoring in adaptive load sharing. In *Proceedings of the Fourth International Conference on Autonomic Computing (ICAC)*, 2007.

[14] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jorg Sander. Lof: Identifying density-based local outliers. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Dalles, TX, 2000.

[15] Mark Burgess. Probabilistic anomaly detection in distributed computer networks. *Science of Computer Programming*, 60:1–26, 2006.

[16] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *Proceedings of SIAM Conference on Data Mining*, 2006.

[17] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2003.

[18] 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.

[19] David M. Chess, Alla Segal, Ian Whalley, and Steve R. White. Unity: Experiences with a prototype autonomic computing system. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 140–147, 2004.

[20] George Cybenko and Vincent H. Berk. Process query systems. *Computer*, 40(1):62–70, 2007.

[21] Mads Dam, Gunnar Karlsson, Babak Firozabadi, and Rolf Stadler. A research agenda for distributed policy-based management. In *Proceedings of nd SEAS DTC Technical Conference*, 2007.

[22] 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.

[23] Paul deGrandis and Giuseppe Valetto. Elicitation and utilization of application-level utility functions. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2009.

[24] Niels Drost, Rob V. van Nieuwpoort, Jason Maassen, and Henri E. Bal. Resource tracking in parallel and distributed applications. In *Proceedings of the Conference on High Performance Distributed Computing (HPDC)*, 2008.

[25] Ivana Dusparic and Vinny Cahill. Using distributed w-learning for multi-policy optimization in decentralized autonomic systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2009.

[26] Martin Ester, Hans-Peter Kriegel, Joerg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*, 1996.

[27] 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.

[28] Dror Feitelson. The parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[29] Thomas Gamer, Michael Scharf, and Marcus Scholler. Collaborative anomaly-based attack detection. *LNCS: Self-Organizing Systems*, 4725:280–287, 2007.

[30] K. C. Gowda and G. Krishna. Agglomerative clustering using the concept of mutual nearest neighborhood. *Pattern Recognition*, 10:105–112, 1977.

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

[32] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1998.

[33] Dan Gunter, Brian L. Tierney, Aaron Brown, Martin Swany, John Bresnahan, and Jennifer M. Schopf. Log summarization and anomaly detection for troubleshooting distributed systems. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (GRID)*, 2007.

[34] A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

[35] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[36] 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.

[37] 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.

[38] Guofei Jiang, Haifeng Chen, Kenji Yoshihira, and Akhilesh Saxena. Ranking the importance of alerts for problem determination in large computer systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2009.

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

[40] Klaus Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security*, 6(4):443–471, 2003.

[41] George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: A hierarchical clustering algorithm using dynamic modeling. *IEEE Computer*, 32(8):68–75, 1999.

[42] Jeffrey Kephart and Rajarshi Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[43] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[44] Omid Khalili, Jiahua He, Catherine Olschanowsky, Allan Snavely, and Henri Casanova. Measuring the peformance and reliability of production computational grids. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID)*, 2006.

[45] 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.

[46] Torsten Klie, Benjamin Ernst, and Lars Wolf. Automatic policy refinement using owl-s and semantic infrastructure information. In *Proceedings of MACE Workshop*, 2007.

[47] Cristopher Krugel, Thomas Toth, and Clemens Kerer. Decentralized event correlation for intrusion detection. In *Proceedings of the International Conference on Information Security and Cryptology (ICISC)*, 2001.

[48] Dara Kusic and Nagarajan Kandasamy. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. *Cluster Computing*, 10(4):395–408, 2007.

[49] Mei Li, Wang-Chien Lee, and Anand Sivasubramaniam. Pens: An algorithm for density-based clustering in peer-to-peer systems. In *Proceedings of the International Conference on Scalable Information Systems (INFOSCALE)*, June 2006.

[50] G.J.A. Loeven, J.A.S. Witteveen, and H. Bijl. Probabilistic collocation: An efficient non-intrusive approach for arbitrarily distributed parametric uncertainties. In *Proceedings of the 45th AIAA Aerospace Sciences Meeting and Exhibit*, 2007.

[51] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25:852–869, 1999.

[52] J. Mao and A.K. Jain. A self-organizing network for hyperellipsoidal clustering (hec). *IEEE Transactions on Neural Networks*, 7:16–29, 1996.

[53] Jonathan Moffet and Morris Sloman. Policy hierarchies for distributed systems management. *IEEE Journal on Selected Areas in Communications*, 11:1404–1414, 1993.

[54] Raffaele Montella and Giuseppe Agrillo. A globus toolkit 4 based instrument service for environmental data acquisition and distribution. In *Proceedings of the Conference on High Performance Distributed Computing (HPDC) - UPGRADE-CN Workshop*, 2008.

[55] Bongki Moon, H.V. Jagadish, and Christos Faloutsos. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, Jan/Feb 2001.

[56] Raymond H. Myers, Douglas C. Montgomery, and Christine M. Anderson-Cook. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. John Wiley and Sons, 2009.

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

[58] Ramon Nou, Ferran Julia, David Carrera, Kevin Hogan, Jordi Caubet, Jesus Labarta, and Jordi Torres. Monitoring and analyzing a grid middleware node. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID)*, 2006.

[59] Elth Ogston, Benno Overeinder, Maarten van Steen, and Frances Brazier. A method for decentralized clustering in large multi-agent systems. In *Proceedings of the 2nd International Joint Conference on Autonomous Agent and Multi-Agent Systems*, pages 798–796, 2003.

[60] Simon Ostermann, Alexandru Iosup, Nezih Yigibasi, Radu Prodan, Thomas Fahringer, and Dick Epema. An early performance analysis of cloud computing services for scientific computing. Technical report, Delft University of Technology, December 2008.

[61] Shrideep Pallickara, Marlon Pierce, Harshawardhan Gadgil, Geoffrey Fox, Yan Yan, and Yi Huang. A framework for secure end-to-end delivery of messages in publish/subscribe systems. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID)*, 2006.

[62] Jaehong Park and Ravi Sandhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the Symposium on Access Control Models and Technologies*, pages 57–64, 2002.

[63] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[64] Wei Peng, Tao Li, and Sheng Ma. Mining logs for computing system management. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*, 2005.

[65] Phillip A. Porras and Peter G. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, 1997.

[66] Leonid Portnoy, Eleazar Eskin, and Sal Stolfo. Intrusion detection with unlabeled data using clustering. In *Proceedings of the ACM CSS Workshop on Data Mining Applied to Security (DMSA)*, 2001.

[67] Andres Quiroz, Nathan Gnanasambandam, Manish Parashar, and Naveen Sharma. Robust clustering analysis for the management of self-monitoring distributed systems. *Journal of Cluster Computing*, Online:–, 2008.

[68] Andres Quiroz, Hyunjoo Kim, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (GRID)*, 2009.

[69] Andres Quiroz, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma. Clustering analysis for the management of self-monitoring device networks. In *Proceedings of the International Conference on Autonomic Computing (ICAC 2008)*, Chicago, June 2008.

[70] Lavanya Ramakrishnan and Daniel A. Reed. Performability modeling for scheduling and fault tolerance strategies for scientific workflows. In *Proceedings of the Conference on High Performance Distributed Computing (HPDC)*, 2008.

[71] 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.

[72] Christopher Roblee, Vincent Berk, and George Cybenko. Implementing large-scale autonomic server monitoring using process query systems. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*, 2005.

[73] Ivan Rodero, Juan Jaramillo, Andres Quiroz, Manish Parashar, and Francesc Guim. Energy-aware online provisioning for virtualized clouds and data centers. Submitted to the ACM International Symposium on High Performance Distributed Computing (HPDC2010).

[74] 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.

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

[76] 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.

[77] 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.

[78] Lakshmikant Shrinivas and Jeffrey F. Naughton. Issues in applying data mining to grid job failure detection and diagnosis. In *Proceedings of the Conference on High Performance Distributed Computing (HPDC)*, 2008.

[79] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994.

[80] 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.

[81] 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.

[82] John Strassner and Declan O'Sullivan. Knowledge management for context-aware, policy-based ubiquitous computing systems. In *Proceedings of the ICAC Workshop on Managing Ubiquitous Communication and Services*, 2009.

[83] Q. Tang, S. Kumar, S. Gupta, and G. Varsamopoulos. Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1458–1472, 2008.

[84] Q. Tang, K. Sandeep, S. Gupta, D. Stanzione, and P.Cayton. Thermal-aware task scheduling to minimize energy usage of blade server based datacenters. In *Proceedings of the 2nd IEEE International Symposium on Dependable Autonomic and Secure Computing (DASC)*, 2006.

[85] Anand Tripathi, Tanvir Ahmed, Sumedh Pathak, Megan Carney, and Paul Dokas. Paradigms for mobile agent-based monitoring of network systems. In *Networks Operations and Management Symposium*, 2002.

[86] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the Second International Conference on Autonomic Computing*, 2005.

[87] G. Gary Wang and S. Shan. Review of metamodeling techniques in support of engineering design optimization. *ASME Transactions, Journal of Mechanical Design*, 129(4):370–380, April 2007.

[88] Qiang Wang and Megalooikonomou Vasileios. A clustering algorithm for intrusion detection. In *Proceedings of the Conference on Data Mining, Intrusion Detection, Information Assurance and Data Networks Security*, 2005.

[89] Mort Webster, Menner A. Tatang, and Gregory J. McRae. Application of the probabilistic collocation method for an uncertainty analysis of a simple ocean model. Technical report, MIT, 1996.

[90] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 2–9, 2004.

[91] Yongwei Wu, Yulai Yuan, Guangwen Yang, and Weimin Zheng. Load prediction using hybrid model for computational grid. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (GRID)*, 2007.

[92] Guangsen Zhang and Manish Parashar. Cooperative defense against ddos attacks. *Journal of Research and Practice in Information Technology (JRPIT)*, 1:1–12, 2005.

[93] Yuanyuan Zhang, Wei Sun, and Yasushi Inoguchi. Predicting running time of grid tasks based on cpu load predictions. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID)*, 2006.

[94] Ying Zhao, Yongmin Tan, Zhenhuan Gong, Xiaohui Gu, and Mike Wamboldt. Self-correlating predictive information tracking for large-scale production systems.

In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2009.

Curriculum Vitae

Andres Quiroz Hernandez

2010 Ph.D. Candidate in Computer Engineering, GPA 3.92/4.0; Rutgers University, NJ, USA

2007 MS in Computer Engineering, GPA 3.88/4.0; Rutgers University, NJ, USA

2004 BS in Systems Engineering, GPA 4.71/5.0; Eafit University, Medellin, Colombia

2005-2010 Graduate Assistant, Center for Autonomic Computing, Rutgers University, NJ, USA

2006 Summer Intern, ISTC, Xerox Corporation, Webster, NY, USA

2004-2005 Research and Teaching Assistant, Eafit University, Medellin, Colombia

2002-2003 Research Intern, Center for Advanced Information Processing, Rutgers University, NJ, USA

Publications

A. Quiroz, M. Parashar, N. Gnanasambandam, and N. Sharma. Autonomic Policy Adaptation using Decentralized Online Clustering. To appear in *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2010.

A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma. Towards Autonomic Workload Provisioning for Enterprise Grids and Clouds. In *Proceedings of IEEE GRID*, October 2009.

A. Quiroz, N. Gnanasambandam, M. Parashar, and N. Sharma. Robust Clustering Analysis for the Management of Self-Monitoring Distributed Systems. *Cluster Computing Journal*, 12:73-85, 2009.

A. Quiroz, M. Parashar, N. Gnanasambandam, and N. Sharma. Clustering Analysis for the Management of Self-Monitoring Device Networks. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, June 2008.

A. Quiroz and M. Parashar. A Framework for Distributed Content-based Web Services Notification in Grid Systems. *Future Generation Computing Systems*, 24(5):452-459, May 2008.

N. Jiang, A. Quiroz, C. Schmidt, and M. Parashar. Meteor: A Middleware Infrastructure for Content-based Decoupled Interactions in Pervasive Grid Environments. *Concurrency and Computation: Practice and Experience*, 20:1455-1484, November 2007.

A. Quiroz and M. Parashar. Design and Implementation of a Distributed Content-based Notification Broker for WS-Notification. In *Proceedings of IEEE GRID*, September 2006.

A. Quiroz and H. Trefftz. Combinatory Multicast for Differentiated Data Transmission in Distributed Virtual Environments. In *Proceedings of IASTED Computer Graphics and Imaging (CGIM)*, August 2004.

A. Agudelo, L. Escobar, J. Restrepo, A. Quiroz, H. Trefftz. A Collaborative Tool for Synchronous Distance Education. In *Proceedings of Computers and Advanced Technology in Education (CATE)*, August 2004.