A Middleware for Self-Managing Large-Scale Systems

CONSTANTIN M. ADAM

Doctoral Thesis
School of Electrical Engineering
KTH Royal Institute of Technology
Stockholm, Sweden 2006
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges
till offentlig granskning för avläggande av Ph.D. December 1, 2006 10.00AM i Sa-
longen, KTHB, Kungl Tekniska Högskolan, Osquarsbacke 31, Stockholm.

© Constantin M. Adam, November 2006

Tryck: Universitetsservice US AB
Abstract

This thesis investigates designs that enable individual components of a distributed system to work together and coordinate their actions towards a common goal. While the basic motivation for our research is to develop engineering principles for large-scale autonomous systems, we address the problem in the context of resource management in server clusters that provide web services.

To this end, we have developed, implemented and evaluated a decentralized design for resource management that follows four principles. First, in order to facilitate scalability, each node has only partial knowledge of the system. Second, each node can adapt and change its role at runtime. Third, each node runs a number of local control mechanisms independently and asynchronously from its peers. Fourth, each node dynamically adapts its local configuration in order to optimize a global utility function.

The design includes three fundamental building blocks: overlay construction, request routing and application placement. Overlay construction organizes the cluster nodes into a single dynamic overlay. Request routing directs service requests towards nodes with available resources. Application placement partitions the cluster resources between applications, and dynamically adjusts the allocation in response to changes in external load, node failures, etc.

We have evaluated the design using complexity analysis, simulation and prototype implementation. Using complexity analysis and simulation, we have shown that the system is scalable, operates efficiently in steady state, quickly adapts to external events and allows for effective service differentiation by a system administrator. A prototype has been built using accepted technologies (Java, Tomcat) and evaluated using standard benchmarks (TPC-W and RUBiS). The evaluation results show that the behavior of the prototype matches closely that of the simulated design for key metrics related to adaptability and robustness, therefore validating our design and proving its feasibility.
Acknowledgments

First I would like to thank my advisor Rolf Stadler for his support, for the fruitful conversations, and for introducing me to the world of scientific research. Furthermore, I would like to thank professor Gunnar Karlsson and all the people at LCN for the stimulating atmosphere, which made it possible for me to concentrate my efforts on research.

Additionally I would like to express my gratitude to the organizations and entities that have funded this research, IBM, the Swedish Foundation for Strategic Research and the Graduate School in Telecommunications at KTH.

I would like to express my special thanks to my wife Shiloh, my parents Sanda and George and the rest of my family and friends for their understanding, patience, and continuous support.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Problem</td>
<td>3</td>
</tr>
<tr>
<td>1.3 The Approach</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Contribution of this Thesis</td>
<td>6</td>
</tr>
<tr>
<td>2 Related Research</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Centralized Management of Web Services</td>
<td>9</td>
</tr>
<tr>
<td>2.2 (Un)Structured Peer-to-Peer Systems in Support of Application Services</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Epidemic Protocols for Constructing Overlays</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Job Scheduling in Grid Computing</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Utility Functions to Control System Behavior</td>
<td>13</td>
</tr>
<tr>
<td>2.6 Application Placement in Distributed Systems</td>
<td>14</td>
</tr>
<tr>
<td>3 Summary of Original Work</td>
<td>17</td>
</tr>
<tr>
<td>4 Open Problems for Future Research</td>
<td>21</td>
</tr>
<tr>
<td>5 List of Publications in the Context of this Thesis</td>
<td>23</td>
</tr>
<tr>
<td>6 Externally Controllable, Self-Organizing Server Clusters</td>
<td>25</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>26</td>
</tr>
<tr>
<td>6.2 System Design</td>
<td>29</td>
</tr>
<tr>
<td>6.3 System Evaluation through Simulation</td>
<td>32</td>
</tr>
<tr>
<td>6.4 External Control and Monitoring of the System</td>
<td>39</td>
</tr>
<tr>
<td>6.5 Related Work</td>
<td>44</td>
</tr>
<tr>
<td>6.6 Discussion and Future Work</td>
<td>48</td>
</tr>
</tbody>
</table>
CONTENTS

7 A Middleware Design for Large-scale Clusters Offering Multiple Services 51
  7.1 Introduction ............................................. 51
  7.2 System Design ........................................... 54
  7.3 System Evaluation Through Simulation ......................... 61
  7.4 Discussion of the Results ................................ 65
  7.5 Related Work ............................................ 69
  7.6 Discussion and Future Work ............................... 72

8 Implementation and Evaluation of a Middleware for Self-Organizing Decentralized Web Services 75
  8.1 Introduction ............................................. 75
  8.2 Overview of the Chameleon Middleware Design ............... 76
  8.3 Implementation: Integrating Chameleon into the Tomcat Framework 80
  8.4 System Evaluation ....................................... 82
  8.5 Related Work ............................................ 88
  8.6 Discussion and Future Work ............................... 89

9 A Decentralized Application Placement Controller for Web Applications 91
  9.1 Introduction ............................................. 91
  9.2 Overview of Our P2P Middleware System .................... 93
  9.3 Decentralized Application Placement Controller ........... 94
  9.4 Experimental Results .................................... 99
  9.5 Related Work ............................................ 104
  9.6 Conclusion ................................................ 104

10 A Service Middleware that Scales in System Size and Applications 105
  10.1 Introduction ............................................ 105
  10.2 System Design .......................................... 107
  10.3 System Evaluation through Simulation ....................... 116
  10.4 Related Work ............................................ 125
  10.5 Discussion and Future Work .............................. 125
  10.6 Acknowledgments ......................................... 126

11 Implementation of a Service Middleware that Scales in System Size and Applications 127
  11.1 Implementation .......................................... 127
  11.2 Evaluation ............................................... 129
  11.3 Discussion ............................................... 136

Bibliography 137
Chapter 1

Introduction

1.1 Background and Motivation

This thesis investigates designs that enable the individual components of a distributed system to work together and coordinate their actions towards a common goal. While the basic motivation is to contribute towards engineering principles for large-scale autonomous systems, this research addresses the problem in the context of resource management in clusters that provide web services. Such clusters offer web applications that must scale to a large number of clients and generate customized dynamic content that matches the clients' preferences. Serving dynamic content can require orders of magnitude more processing resources compared with serving purely static data [1]. Consequently, web sites providing mainly dynamic content can experience processing bottlenecks, making it necessary to introduce mechanisms for resource control that manage CPU and memory on their nodes.

Advanced architectures for cluster-based services that have been recently proposed (including commercial solutions like IBM WebSphere [2] or BEA WebLogic [3], and research prototypes like Ninja [4] or Neptune [5]) allow for service differentiation, server overload control and high utilization of resources. These systems, however, rely on centralized resource allocation, which limits their ability to scale and tolerate faults. A centralized resource manager can control a group of several hundreds of nodes. In order to improve further the scalability of this solution, one can bridge multiple groups of nodes into a single large cluster. However, the complexity of configuring and managing such a cluster increases with the number of groups involved (see Fig. 1.1(a)).

Current service networks incorporate tens of thousands of nodes and are expected to further expand in the future. As a point of reference, the content delivery network used today by Akamai contains some 20,000 nodes [6]. As recent research in peer-to-peer systems [7, 8, 9, 10] and distributed management [11, 12, 13]
CHAPTER 1. INTRODUCTION

(a) Bridging several groups of nodes.       (b) Organizing the nodes into a single group using a decentralized design.

Figure 1.1: This thesis investigates designs for large-scale, self-configuring node clusters. The design includes the nodes in data centers, the entry points and a management station.

suggests, a decentralized solution for resource management can potentially eliminate the configuration complexity for resource management in large-scale service networks. Under such a design, all the nodes belong to a single group that can increase in size as much as needed, while its resources are partitioned between the applications offered by the system (as shown in Fig. 1.1(b)).

The core contribution of this thesis is the introduction of self-management capabilities into the design of cluster-based large-scale services. To achieve this, we have developed, implemented and evaluated a decentralized design for resource management in these systems. The design makes service platforms dynamically adapt to the needs of customers and to environmental changes, while giving service providers the capability to adjust operational policies at runtime. The design provides the same functionality as the advanced service architectures mentioned above (i.e., service differentiation, efficient operation and controllability), while, at the same time, assuming key properties of peer-to-peer systems (i.e., scalability and adaptability).

This research is an extension of the author’s licentiate thesis [14], which presented a decentralized design for allocating resources among multiple services inside a server cluster. (In Sweden, a licentiate is an intermediate degree between M.Sc. and Ph.D.). In this work, we improve the scalability of the architecture in [14], extend its functionality, develop an evaluation methodology for the design and thoroughly evaluate the system through complexity analysis, simulation and testbed implementation.
1.2 The Problem

Engineering a resource management system for cluster-based services includes addressing two problems. The first problem, Application Placement, refers to developing an approach for allocating the system resources to a set of applications that the cluster offers by deciding which application should run on which node. The allocation must be efficient, and must take into account the constraints of the resource requirements of the applications, the quality of service objectives of the clients and the policies for service differentiation under overload. The second problem, Request Routing, refers to developing a scheme for directing client requests to available resources inside the cluster. The routing scheme must be efficient and ensure a balanced load in the system.

While providing the functionality described above, the resource management system must satisfy the design goals of scalability, adaptability and manageability. The design goal of scalability means that, while the processing resources of the system increase proportionally with its number of nodes, the control load on a node must remain bounded and must be independent of the system size. The design goal of adaptability means that the configuration overhead introduced by node additions or removals must be minimal, that the system must adapt to external events, such as load changes or node failures, and that the nodes must coordinate their local actions to achieve a global objective. In the context of this work, the design goal of manageability means that an administrator must have the capability to monitor the behavior of the system and to control it through high-level policies. The system then adjusts its configuration in response to changes in management policies.

1.3 The Approach

Design Principles

In order to achieve the design goals listed above, our design of the middleware for resource management follows four principles. The first three of them are characteristic to many peer-to-peer systems. First, in order to facilitate scalability, each node has only partial knowledge of the system, and it receives and processes state information from a subset of peers, the size of which is independent of the system size. Second, each node can adapt and change its role at runtime. Initially, all nodes are functionally identical in the sense that they have the capability to manage their resources locally and to determine which applications they offer. Third, each node runs a number of local control mechanisms independently and asynchronously from its peers. Note that communication among peers does not introduce synchronization overhead, as each node communicates only with one peer at a time. This design feature allows a large system to adapt quickly to external events. As each node runs its local control mechanisms asynchronously and independently from other nodes, (periodic) local control operations are distributed over time, which lets parts of the
systems re-configure shortly after events (such as load changes or node failures) occur.

The fourth design principle is specific to the design of our resource management system: each node adapts its local configuration in order to optimize a global utility function. The system optimizes the utility function in a decentralized fashion using the following heuristic: each node gathers a partial view of the system state from its neighborhood and takes the control decisions (e.g., starting or stopping applications) that maximize the neighborhood utility.

**Decentralized Control Mechanisms**

We have identified the following three decentralized mechanisms (presented in Fig. 1.2) as the building blocks of the design: overlay construction, request routing and application placement.

*Overlay construction*, based on an epidemic protocol [15] and a set of local rules for building the overlay [16], organizes the cluster nodes into a single dynamic overlay that regenerates quickly after failures and can be optimized according to specific criteria.

*Request routing* directs service requests towards nodes with available resources. By selectively propagating routing updates, this mechanism is capable of bounding
the processing load on a node to a constant value that is independent of the system size.

Application placement partitions the cluster resources between services, in response to changes in external load, updates of the control parameters, or node failures. The placement process is driven by a global objective that we model using a utility function.

System Evaluation

The design is evaluated according to five criteria: efficiency, scalability, adaptability, robustness and manageability. Efficiency is the capability of the system to exhibit high performance in steady state. Scalability is the capability of the system to increase its processing resources proportionally with its number of nodes, while the control load on a node remains bounded and independent of the system size. Adaptability is the capability of the system to respond to a change in the operating conditions by reconfiguring and converging to a new steady state. Robustness is the capability of the system to respond to node arrivals, departures and failures, by reconfiguring and converging to a new steady state. In this work, we understand manageability as the capability of the system to adjust its configuration to changes in management policies.

We use three evaluation methods: complexity analysis, simulation and testbed implementation.

Iterative Development of the Design

The development of the design has been an iterative process. This thesis presents three decentralized architectures for managing resources in clusters that provide web services. Each version of the architecture can be seen as an improvement and an extension of the previous version. The description of the architectures illustrates the entire development path of the concepts described here. All three architectures use the same type of control mechanisms (overlay construction, request routing and application placement) as fundamental building blocks. While these types of mechanisms remain the same, their structure changes and their functionality increases in each architecture.

The first architecture (Chapter 6) addresses the problem of a system that provides a single application. The three decentralized control mechanisms combined achieve scalability in terms of system size, and they achieve robustness to node failures. Overlay construction maintains a dynamic overlay of cluster nodes. Request routing directs service requests towards servers that are not overloaded. Membership control allocates/releases servers to/from the cluster, in response to changes in the external load.

The second architecture (Chapters 7, 8) introduces support for multiple applications in the system and allows an administrator to express a global objective (e.g., quality of service objectives for each application and service differentiation under
overload) through utility functions. A key feature of this architecture (included
in the application placement mechanism) is the methodology of optimizing utility
functions in a decentralized fashion.

The third architecture (Chapters 9, 10) provides support for several applications
to run concurrently on the same node. In addition, this architecture has the capa-
bility to manage several types of resources (e.g., CPU and memory). The design of
each of the three decentralized control mechanisms has been improved compared to
the previous architectures. Each mechanism runs on a different timescale to achieve
its own objective.

In this architecture, overlay construction uses an epidemic algorithm and a set of
local rules to build the overlay. Compared to the previous architecture, the structure
of the overlay has changed from a unidirectional dynamic graph to a bidirectional
stable graph, in which each node has approximately \( \pm 1 \) the same number of
neighbors. The topological properties of such an overlay facilitate balancing the
control load. They also simplify unbiased estimation of the system state from a
subset of nodes, as the state of each node can be sampled the same number of times
by other nodes, and each node has a neighborhood of the same size for retrieving
state information.

Request routing uses a novel scheme for selective propagation of updates, which
allows each node to locate an application provider in a single step. For a bounded
number of applications, the scheme limits the processing load on a node to a con-
stant value that is independent of the system size. Moreover, this scheme allows
each application provider to be present in approximately the same number of rout-
ing tables, a feature that is desirable for load balancing.

Application placement uses a simple model for service differentiation, where each
application is assigned an importance factor. Increasing the value of the importance
factor of a specific application results in its deployment on a larger number of nodes.
As in the previous architecture, the functionality of this mechanism is based on the
decentralized optimization of a utility function.

1.4 Contribution of this Thesis

A Scalable and Robust Design with Low Complexity

This thesis presents a novel decentralized design that performs resource manage-
ment in large-scale clusters offering web services. We provide experimental proof
that with only three key decentralized mechanisms (overlay construction, request
routing and application placement) one can build a middleware for web services
that is scalable, adaptable and manageable.

Selective Propagation of Routing Updates

Request routing uses a novel scheme for selective propagation of updates that is
based on logical proximity. The basic idea is that a node maintains information
1.4. \textit{CONTRIBUTION OF THIS THESIS}

about a fixed number of providers for each application. Upon receiving a routing update, the node propagates it further only if it causes a modification to its routing table. This selective dissemination of routing updates restricts the control load on a node to a value that is independent of the system size (for a fixed number of applications). Moreover, this scheme allows each application provider to be present in approximately the same number of routing tables, a feature that is desirable for load balancing.

\textbf{Utility Functions to Express and Achieve Global Objectives}

We use cluster utility functions to define a global objective for the system and to measure how well the system meets the objective. The cluster utility function represents a composition of several application utility functions.

The exact formulas for the application utility functions and the cluster utility function depend on the global objective of the system. For example, a system where the cluster utility function is defined as the minimum of the application utility functions provides fair allocation to all cluster applications. A system where the cluster utility function is defined as the sum of the application utility functions allocates the cluster resources to applications in such a way that the performance targets of applications with higher values for control parameters are more likely to be met than those of services with lower values. This enables service differentiation in case of overload.

In the context of this thesis, we have applied two types of application utility functions. The first such function allows the association of two performance targets with each application: the maximum response time (defined per individual request) and the maximum drop rate (defined over all the requests for a specific service). The drop rate for an application represents the ratio between the number of requests served under response time constraints and the total number of incoming requests for the application. The application utility function specifies the rewards for meeting and the penalties for missing the performance targets for that application. Each application function has two control parameters, $\alpha$ and $\beta$, which define the shape of the function graph and determine the relative importance of an application.

The second application utility function used in this thesis assigns to each application $a$ an importance factor $w_a$. Applications with higher values for the importance factors will be deployed on a larger number of nodes. The utility provided by a node is the weighted sum of the CPU resources the node supplies to each application.

\textbf{System Implementation and Evaluation}

We have evaluated the design using complexity analysis, simulation and prototype implementation. We have used complexity analysis to determine the messaging load generated by each of the control mechanisms running on a node. The analysis shows that the control load per node increases linearly with the number of applications,
but is independent of the system size. The simulation results show that the system is scalable within the parameter range tested in the scenarios. The process-based simulation [17] ensures that the simulation model is close to a real implementation. Finally, the measurements from the implementation on the testbed show that, based on our design, an efficient, adaptable, robust and manageable prototype can be built. The prototype has been constructed using accepted technologies (Java [18], Tomcat [19]) and has been evaluated using standard benchmarks (TPC-W [20] and RUBS [21]).

We have extended the design to allow for managing the system from an external management station. We have added schemes for disseminating control parameters and estimating state variables. The management station can contact any active server in the cluster for changing a management parameter in the entire system or for reading an estimate of a global performance metric. We have demonstrated the reaction of the system to changes in quality of service policies and its capability to monitor, at runtime, global performance parameters.

**Published Work**

The work presented in this thesis has lead to thirteen papers, out of which eleven have been published in conferences or journals (see Chapter 5).
Chapter 2

Related Research

Various aspects of our research relate to platforms for web services with quality of service objectives, peer-to-peer systems, applications of epidemic protocols, activities within the grid computing community and systems controlled by utility functions.

2.1 Centralized Management of Web Services

Today’s server clusters are engineered following a three-tiered architecture. Service requests enter the cluster through one or several layer-4/7 switches that dispatch them to the servers inside the cluster. In such architecture, the functionality of layer-4/7 switches has been enhanced to provide firewall functionality, anti-virus screening and load-balancing in the cluster. Layer-4/7 switches offer a combination of Network Address Translation (NAT) and higher-layer address screening ([22, 23]). Generally, they make forwarding decisions based upon information at OSI layers 4 through 7. Some layer-4/7 switches, also called session switches, are capable of monitoring the state of individual sessions. This functionality enables them to balance traffic across a cluster of servers, based upon individual session information and status.

Many recent works focus on the problem of efficiently dispatching requests into the server cluster. In [24], a scheduler with dynamic weights which controls overload in web servers is proposed. The weighted-fair queue scheduler proposed in [25] uses an observation-based adaptive scheme to increase the weight of a service class experiencing poor performance at the expense of another class that has more resource share and less demand. In [26], the authors compare various locally distributed web system architectures and give a classification and analysis of various dispatching algorithms.

While layer-4/7 switches can provide load balancing functionality, their design becomes increasingly complex for clusters where the number of servers, applications and quality of service objectives becomes large. Moreover, if several layer-4/7
switches forward requests inside a server cluster, they need to synchronize their knowledge about the server states.

In [27], a performance management system for cluster-based web services is presented. The system dynamically allocates resources to competing services, balances the load across servers, and protects servers against overload. The system described in [28] adaptively provisions resources in a hosting center to ensure efficient use of power and server resources. The system attempts to allocate dynamically to each service the minimal resources needed for acceptable service quality; it leaves surplus resources available to deploy elsewhere. In [29], the authors present an architecture in which dispatchers at an overloaded Internet data center (IDC) redirect requests to a geographically remote but less loaded IDC. Even though requests are routed between several IDCs, we still argue that this is a centralized scheme, as the request dispatchers and the servers are organized in a hierarchical centralized structure inside each IDC.

The cluster architecture in [27] contains several types of components that share monitoring and control information via a publish/subscribe network. Servers and gateways continuously gather statistics about incoming requests and send them periodically to the Global Resource Manager (GRM). GRM runs a linear optimization algorithm that takes as input the statistics from the gateways and servers, the performance objectives, the cluster utility function, and the resource configuration. GRM computes two parameters: the maximum number of concurrent requests that server $s$ executes on behalf of the gateway $g$ and the minimum number of class $c$ requests that every server executes on the behalf of each gateway. GRM then forwards the new parameter values to the gateways, which apply them until they receive a new update.

Similarly, the Muse resource management scheme ([28]) contains several types of components: servers, programmable network switches, and the executor. Servers gather statistics about incoming requests and process assigned requests. The programmable switches redirect requests towards servers following a specific pattern. Finally, the executor periodically computes an optimal resource allocation policy, which takes as input the bids for services from customers on one side, and the service statistics from servers on the other side.

As in our design, both approaches described in [27] and [28] map service requests into service classes, whereby all requests in a service class have the same quality of service objective.

Two main characteristics distinguish the design presented in this thesis from these two approaches: our design is decentralized, and all our cluster components are of the same type. We believe that our approach leads to a lower system complexity and, thus, the task of configuring the system becomes simpler. In addition, it eliminates the single point of failure, namely, GRM in [27] and the executor in [28].

In [30], the authors present a technique for fragment-based web caching, which relies on an algorithm [31] for detecting fragments in dynamic web pages and also on a standard [32] for fragment-based publishing, caching and delivery of web data.
2.2. (Un)Structured Peer-to-Peer Systems in Support of Application Services

The fragmenting approach increases cacheable web content, decreases data invalidation and aids efficient utilization of disk space. The authors identify the mechanisms that provide quality of service for dynamic web content: mechanisms that detect overload ([33, 34, 35, 36, 37]), mechanisms that react to overload ([38, 39]) and mechanisms for admission control ([37, 40, 41]). None of these mechanisms, however, address the quality of service problem in large-scale, distributed settings, as our work does.

Studying the interactions between the components of multi-tier systems is another research topic in the area of quality of service for Web services. In [42], the authors develop an analytical model for multi-tier Internet services. They propose using this model for tasks such as capacity provisioning, performance prediction, application configuration or request policing. In other works ([43], [44]), the authors propose mechanisms that prevent overload and saturation of the database servers in small-sized clusters, by controlling the interaction between servlets and the database server. Finally, application profiling, as presented in ([45], [46], [47]) represents an important activity needed for engineering web services with quality of service objectives.

2.2 (Un)Structured Peer-to-Peer Systems in Support of Application Services

The design in this thesis shares several principles with peer-to-peer systems. After having studied the possibility of developing our architecture on top of a structured peer-to-peer system, we concluded that such an approach would likely lead to a system that is more complex and less efficient than the one presented in this thesis, and we explain here briefly why. (To keep the term short, we use peer-to-peer system instead of structured peer-to-peer system.) Peer-to-peer systems are application-layer overlays built on top of the Internet infrastructure. They generally use distributed hash tables (DHTs) to identify nodes and objects, which are assigned to nodes. A hash function maps strings that refer objects to a one-dimensional identifier space, usually the interval \([0, 2^{128} - 1]\). The primary service of a peer-to-peer system is to route a request with an object identifier to a node that is responsible for that object. Routing is based on the object’s identifier and most systems perform routing within \(O(\log n)\) hops, where \(n\) denotes the system size. Routing information is maintained in the form of a distributed indexing topology, such as a circle or a hypercube, which defines the topology of the overlay network.

Even though peer-to-peer networks efficiently run best-effort services ([48], [49], [50], [51]), no results are available to date on how to achieve service guarantees and service differentiation using peer-to-peer middleware. If one wanted to use a peer-to-peer layer as part of the design of a server cluster, one would assign an identifier to each incoming request and then assign the peer-to-peer system route the request to the node responsible for that identifier. The node would then process the request. In order for the server cluster to efficiently support quality of service...
objectives, some form of resource control or load balancing mechanism would be
needed in the peer-to-peer layer.

Introducing load-balancing capabilities in DHT-based systems is a topic of on-
go ing research ([52, 53, 54, 55]). An interesting result is that uniform hashing by
itself does not achieve effective load balancing. In [52], the authors show that, in
a network with n nodes, where each node covers on average a fraction of 1/n of
the identifier space, with high probability, at least one node will cover a fraction of
O(log n/n) of the identifier space. Therefore, uniform hashing results in an O(log n)
imbalance in the number of objects assigned to a node. Recently proposed solutions
to the problem of load balancing in DHT systems include "the power of two choices"
[52], load stealing schemes, and schemes that include virtual servers ([54, 55]).

In order to implement an efficient resource allocation policy that dynamically
adapts to external load conditions, the identifier space in a peer-to-peer system
needs to be re-partitioned and the partitions reallocated on a continuous basis.
This means that the indexing topology, a global distributed state, needs to be
updated continuously to enable the routing of requests to nodes.

When comparing the overhead associated with request routing based on DHTs
with our routing mechanism, we concluded that maintaining a global indexing topol-
ogy is significantly more complex than maintaining the local neighborhood tables
in our design.

In addition, peer-to-peer systems have properties that are not needed for our
purposes. For instance, a peer-to-peer system routes a request for an object to a
particular server, the one that is responsible for that object. (This property is useful
to implement information systems on peer-to-peer middleware.) In our design, a
request can be directed to any server with available capacity, which simplifies the
routing problem.

2.3 Epidemic Protocols for Constructing Overlays

Epidemic algorithms disseminate information in large-scale systems in a robust and
scalable way. In epidemic algorithms, a node sends local information to one or more
neighbors in an asynchronous way.

The use of epidemic protocols for building overlays has been proposed in the
context of applications such as data aggregation, resource discovery and monitoring
[13], database replication [56, 57] and handling web hotspots [58]. For example,
Astrolabe [13] uses an epidemic protocol for disseminating information and for
building an aggregation tree, which mirrors the administrative hierarchy of the
system.

In the designs presented in this thesis, we apply two epidemic algorithms, News-
cast [59] and CYCLON [15], to locate available resources in a system that is subject
to rapid changes. In our design, the overlay construction mechanism uses Newscast
or CYCLON to construct and maintain the overlay, through which requests are
being routed. We further use Newscast and CYCLON as a basis for disseminating
control parameters and as part of an aggregation scheme to estimate global state variables.

2.4 Job Scheduling in Grid Computing

As in our system, a grid locates available resources in a network of nodes for processing requests (or tasks). Resource management and task scheduling in distributed environments are fundamental topics of grid computing. The goal is often to maximize a given workload, subject to a set of quality of service constraints. Many activities ([60], [61], [62], [63]) focus on statistical prediction models for resource availability. These models are used as part of centralized scheduling architectures, in which a single scheduler dispatches tasks to a set of available machines. Other research ([64]) creates peer-to-peer desktop grids that, however, do not work under quality of service constraints.

More recent work in the Grid Computing community addresses the issue of managing grid resources in a decentralized fashion. In [65], the authors present a middleware that uses JXTA and runs on top of a bidirectional logical overlay.

The grid computing environment has different requirements from our system. The tasks have typically longer execution times, compared to our environment. Nodes can become unavailable and leave the grid in the middle of executing a task. This requires different handling of failures and reconfigurations. For instance, the state of a task on a node should be saved and sent periodically to a neighbor. We see the potential of applying elements of our design to task scheduling in grid computing. The potential benefits of our approach are increased scalability and simplified configuration.

2.5 Utility Functions to Control System Behavior

The idea of controlling the behavior of a computer system using utility functions has been explored in a wide variety of fields: scheduling in real-time operating systems, bandwidth provisioning and service deployment in service overlay networks, design of server clusters and intelligent systems. When applying utility functions, a system can switch to a set of new states, where each state is associated with a reward.

Time utility functions (TUF) have been defined in the context of task scheduling in real-time operating systems [66]. Given a set of tasks, each with its own TUF, the goal is to design scheduling algorithms that maximize the accrued utility from all of the tasks. If task $T$ starts at time $I$ and has a deadline $D$, TUF functions measure the benefit of completing $T$ at different times. For unimodal functions, the benefit decreases over time. For a hard time constraint, the benefit is constant until the deadline $D$, and is zero afterwards. For soft time constraints, the benefit starts decreasing at some time after $I$, and becomes zero at time $D$. In [67], the authors present the design of a real-time switched Ethernet that uses time utility functions.
The authors extend the TUF concept in [66] by defining progressive utility functions and joint utility functions. Progressive utility functions describe the utility of an activity as a function of its progress (e.g. the computational accuracy of the activity’s results). Joint utility functions specify the utility of an activity in terms of completion times of other activities and their own progress.

The bandwidth-provisioning problem for a service overlay network (SON) has been modeled as an optimization based on utility functions, for example in [68]. The problem is to determine the link capacities that can support quality of service sensitive traffic for any source-destination pair in the network, while minimizing the total provisioning costs.

In [69], the authors propose a decentralized scheme for replicating services along an already established service delivery tree. This algorithm estimates a utility function in a decentralized way. However, this process takes place along a fixed service delivery tree. The task of setting up the tree is outside of the scope of the work. Once the service delivery tree is in place, the authors propose an algorithm in which each node interacts with its parent and children within the tree and, as a result, finds a placement of service replicas along the tree that maximizes the total throughput and minimizes the quality of service violation penalty along the tree.

Machine learning is an area of artificial intelligence concerned with the development of techniques that allow computers to interact with the surrounding environment, given an observation of the world [70]. In many works, reinforcement learning algorithms use utility functions to model (a) the impact that every action of a system component has on the surrounding environment and (b) the feedback provided by the environment that guides the learning algorithm.

The systems discussed earlier under the centralized web service architectures used utility functions to partition resources between several classes of service. Apart from [69], we could not find any system design in the literature that includes a decentralized evaluation of a utility function. Evaluating the utility in a decentralized way is one of the key parts of our design.

2.6 Application Placement in Distributed Systems

The dynamic application placement problem, as it is approached in this thesis, is a variant of the class constrained multiple-knapsack problem, which is NP-hard [71]. The application placement problem has been studied extensively in several contexts, and many approaches to solve this problem have been developed. In this section, we review related work in the areas of web services, content delivery networks, stream processing systems, utility computing, and grid computing.

Application Placement for Web Services

The work on application placement in this thesis is closely related to [72]. The fundamental difference is that our approach is decentralized, while the approach in
2.6. APPLICATION PLACEMENT IN DISTRIBUTED SYSTEMS

[72] is centralized. Because of its decentralized nature, our work addresses several points not covered in [72], including continuously adapting the system configuration in response to external events and handling large delays that occur when starting or stopping applications. The modest loss in efficiency due to the decentralized nature of the system is compensated for by a gain in scalability.

Stewart et al. [45] present a method for component placement that maximizes the overall system throughput. The authors consider three types of resources: CPU, memory, and network. Their method has three phases. In the first phase, per-component resource profiles are built. In the second phase, components are placed where they yield high overall throughput. In the third phase, components migrate at runtime, following changes in external conditions. While the authors mention the advantages of a decentralized placement controller, they do not propose such a design.

Some placement controllers [73] allocate entire servers to a single application. By contrast, our application placement algorithm can allocate several applications to share a single server. As shown in [74], fine-grained resource allocation on a short timescale can lead to substantial multiplexing gains.

Content Delivery Networks and Stream Processing Systems

The work in content delivery and stream processing [75, 76, 77] addresses the problem of placing a set of document replicas or stream operators in a network. The placement goals are: (a) minimizing delays by placing the replicas/operators as close to the clients as possible and (b) minimizing the bandwidth used to send the information to the clients. These two goals are conflicting. In [75, 76], the authors define utility functions that assign a cost to the consumed net bandwidth, and a revenue to the quality (low delay) of the service. Maximizing these utility functions yields an optimal placement of the replicas in the network. In [75], the authors describe and analyze a centralized algorithm that works for static systems. In [76], the placement procedure for replicating services is decentralized and it takes place along a given service delivery tree.

In [77], the authors propose an interesting decentralized solution to the operator placement problem. They use two mechanisms: (a) a cost space (using the Vivaldi algorithm [78]), which is a metric space that captures the cost for routing data between nodes, and (b) a relaxation placement algorithm, which places operators using a spring relaxation technique that manages single and multi-query optimization.

Utility Computing

Resource management is a central aspect in utility computing, and many approaches are being developed to support this functionality. Quartermaster [79], for instance, is a set of tools that ensure a consistent system configuration, that dynamically provision resources for applications and that optimize the usage of these
resources at runtime. The Quartermaster tools are integrated using the Common Information Model (CIM). The resource management functionality of Quartermaster enables it to adapt to external events and optimize resource usage dynamically. The main difference between Quartermaster and our work is that Quartermaster follows a centralized architecture, while our approach is decentralized.

Application placement in data centers has been analyzed in the context of utility computing in [80] and [81]. In [80], the authors build a model for the computing and networking components of an Internet data center in which several multi-tier applications are deployed. A number of techniques (including projection of the solution set, partition of the network and pruning of the search space and local clustering for large problems) assign the resources of the data center in such a way that the communication delay between servers is minimized. In [81], the solution to the resource assignment problem for an Internet data center is extended to minimize not only the communication delays, but also the traffic-weighted average inter-server distance. The optimization is subject to the constraints of satisfying the application requirements regarding processing, communication and storage, without exceeding the network capacity limits. The resource allocation problem is formulated as a nonlinear combinatorial optimization problem and three solutions based on different linearization techniques are proposed.

In [82], the authors address the problem of dynamically allocating virtualized resources using feedback control. They present a workload management tool that dynamically controls resource allocation to a hosted application in order to achieve quality of service goals. They propose a feedback-control system consisting of two nested control loops for managing the quality of service metric of the application, along with the utilization of the allocated CPU resource.

In [83] the author presents a decentralized placement algorithm for allocating computational resources on demand in utility data centers. In the author’s framework, each application has several components and the placement algorithm attempts to minimize the distance between application components that exchange large amounts of data. The approach in [83] differs from ours in several ways. First, our design calls for an identical placement controller on each node, while in [83] each application has its own centralized Service Manager. Second, in [83], a set of additional placement components, named Placement Managers, trigger the trading rounds between the Service Managers and maintain two centralized matrices with information about the distance and the maximum bandwidth requirements between components. Finally, in [83], the configuration of a node changes following a resource swap between two Service Managers, while, in our design, each node decides on changing its configuration locally.
Chapter 3

Summary of Original Work

The work presented in this thesis has lead to thirteen papers, out of which eleven have been published (see Chapter 5). Five of these papers have been included in the present thesis.

**Paper A: Externally Controllable, Self-Organizing Server Clusters**

We present a decentralized design for a server cluster that supports a single service with response time guarantees. Three distributed mechanisms represent the key elements of our design. Topology construction maintains a dynamic overlay of cluster nodes. Request routing directs service requests towards available servers. Membership control allocates/releases servers to/from the cluster, in response to changes in the external load. We advocate a decentralized approach, because it is scalable, fault-tolerant, and has a lower configuration complexity than a centralized solution. We demonstrate through simulations that our system operates efficiently by comparing it to an ideal centralized system. In addition, we show that our system rapidly adapts to changing load. We found that the interaction of the various mechanisms in the system leads to desirable global properties. More precisely, for a fixed connectivity $c$ (i.e., the number of neighbors of a node in the overlay), the average experienced delay in the cluster is independent of the external load. In addition, increasing $c$ increases the average delay but decreases the system size for a given load. Consequently, the cluster administrator can use $c$ as a management parameter that permits control of the tradeoff between a small system size and a small experienced delay for the service. Furthermore, we investigate the capabilities of the system to self-organize and effectively adapt to changing load and failures, even massive ones. We demonstrate the reaction of the system to a change in a quality of service policy and its capability to monitor, at runtime, global performance parameters including the average response time and the system size.

This is an extension of the paper “Adaptable Server Clusters with QoS Objectives”, published in the Proceedings of the 9th IFIP/IEEE International Symposium
on Integrated Network Management (IM-2005), Nice, France, May 16-19, 2005. This paper appears in the thesis as Chapter 6.

Paper B: A Middleware Design for Large-scale Clusters Offering Multiple Services

We present a decentralized design that dynamically allocates resources to multiple services inside a global server cluster. The design supports quality of service objectives (maximum response time and maximum loss rate) for each service. A system administrator can modify policies that assign relative importance to services and, in this way, control the resource allocation process. Distinctive features of our design are the use of an epidemic protocol to disseminate state and control information, as well as the decentralized evaluation of utility functions to control resource partitioning among services. Simulation results show that the system operates both effectively and efficiently; it meets the quality of service objectives and dynamically adapts to load changes and to failures. In case of overload, the service quality degrades gracefully, controlled by the cluster policies.

This paper has been published in the IEEE Transactions on Network and Service Management (eTNSM), Vol. 3, No. 1, 2006. This paper appears in the thesis as Chapter 7.

Paper C: Implementation and Evaluation of a Middleware for Self-Organizing Decentralized Web Services

We present the implementation of Chameleon, a peer-to-peer middleware for self-organizing web services, and we provide evaluation results from a test bed. The novel aspect of Chameleon is that key functions, including resource allocation, are decentralized, which facilitates scalability and robustness of the overall system. Chameleon is implemented in Java on the Tomcat web server environment. The implementation is non-intrusive in the sense that it does not require code modifications in Tomcat or in the underlying operating system. We evaluate the system by running the TPC-W benchmark. We show that the middleware dynamically and effectively reconfigures in response to changes in load patterns and server failures, while enforcing operating policies, namely, quality of service objectives and service differentiation under overload.

This paper has been published in the Proceedings of the Second IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006), Dublin, Ireland, June 16, 2006. This paper appears in the thesis as Chapter 8.

Candidate’s Contribution to the Papers D and E

The Paper D is a technical report and the Paper E is a paper that will be published in IM-2007, which contain co-Authors from IBM Research. The role of the IBM researchers was to give comments to problem motivation and the results. The
problem statement, the technical work and the writing was done by Constantin Adam.

**Paper D: A Decentralized Application Placement Controller for Web Applications**

This paper addresses the problem of dynamic system reconfiguration and resource sharing for a set of applications in large-scale services environments. It presents a decentralized application placement scheme that dynamically provisions enterprise applications with heterogeneous resource requirements. Potential benefits, including improved scalability, resilience, and continuous adaptation to external events, motivate a decentralized approach. In our design, all nodes run a placement controller independently and asynchronously, which periodically reallocates a node's local resources to applications based on state information from a fixed number of neighbors. Compared with a centralized solution, our placement scheme incurs no additional synchronization costs. We show through simulations that decentralized placement can achieve accuracy close to that of state-of-the-art centralized placement schemes (within 4% in a specific scenario). In addition, we report results on scalability and transient behavior of the system.

*This paper is available as a Technical Report, IBM Tech. Report RC23980, June 2006. This paper appears in the thesis as Chapter 9.*

**Paper E: A Service Middleware that Scales in System Size and Applications**

We present a peer-to-peer design of a service middleware that dynamically allocates system resources to a large set of applications. The system achieves scalability in number of nodes (1000s or more) through three decentralized mechanisms that run on different time scales. First, overlay construction interconnects all nodes in the system for exchanging control and state information. Second, request routing directs requests to nodes that offer the corresponding applications. Third, application placement controls the set of offered applications on each node, in order to achieve efficient operation and service differentiation. The design supports a large number of applications (100s or more) through selective propagation of configuration information needed for request routing. The control load on a node increases linearly with the number of applications in the system. Service differentiation is achieved through assigning a utility to each application, which influences the application placement process. Simulation studies show that the system operates efficiently for different sizes, adapts fast to load changes and failures and effectively differentiates between different applications under overload.

*This paper will be published in the Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM-2007), Munich, Germany, May 21-25, 2007. This paper appears in the thesis as Chapter 10.*
CHAPTER 3. SUMMARY OF ORIGINAL WORK

Paper F: A Service Middleware that Scales in System Size and Applications

This report describes the implementation on an experimental testbed of the design described in Chapter 10. The implementation is evaluated using the RUBiS benchmark. The results show that the testbed implementation behaves similarly to the simulation, therefore validating our simulation model and proving the feasibility of the design.

This paper contains an implementation of the architecture presented in Paper E and recent measurement results from the testbed which have not been published to date. This paper appears in the thesis as Chapter 11.
Chapter 4

Open Problems for Future Research

There are several issues that need to be addressed in order to further increase the applicability of the current design.

First, we have not explicitly modeled the state of applications. Specifically, we did not address the server affinity problem. Our current design handles all service requests independently of one another and thus has no concept of a session. Many practical applications, though, require a series of requests to be executed within a session. In such a case, one might want to process all of these requests on a single server and keep the session state between requests on the server.

An increasing number of web applications are deployed on a multi-tier architecture, in which each tier uses services provided by its successor tier. For instance, a client request that is processed by a J2EE application, will query a database, incorporate the result of the query into a web page and return that to the client. In order to support such a scenario, a design should address the interactions between the different tiers that support the same application. Such a study would potentially extend the applicability of our design to many e-commerce services or to search engines where the database layer constitutes a separate tier.

The current design does not address the decentralization of the database tier. This issue is a very active research area, and a number of approaches have been proposed (see, for example [84]). Integrating our middleware design with a distributed database technology, such as Oracle RAC [85], and investigating to what extent our design principles can help engineering an effective distributed database tier would be an interesting research challenge.

Our design uses the concept of utility functions to specify a global objective, and we have developed a scheme for continuously maximizing a cluster utility in a decentralized fashion. While we have shown that several types of utility functions (e.g., for supporting quality of service objectives for each application, or for assigning a relative importance to each application) can be applied in the context of server clusters, we did not perform a systematic study that would produce guidelines on how to choose a utility function with desired characteristics.
In the third architecture presented in this thesis (Chapter 10), each of the three decentralized control mechanisms runs on its own timescale. An open question is: What are the appropriate timescales for each of the mechanisms to best achieve an efficient, stable and responsive system? More precisely, what are the timescales on which the nodes refresh their address caches, maintain the overlay topology, measure the load statistics, send routing updates that advertise their configuration and run the application placement algorithm.

A potential application domain for our design is virtualization in large-scale systems. Virtualization technologies ([86], [87]) allow common IT resources, such as processors, networks, storage, and software licenses, to be shared across multiple users, organizations or applications. Our design could be applied to develop technologies that control the virtualization process, e.g., by invoking operations that start, stop or clone virtual machines according to global objectives specified by a system administrator.
Chapter 5

List of Publications in the Context of this Thesis

Chapter 6

Externally Controllable, Self-Organizing Server Clusters

Constantin Adam and Rolf Stadler
Laboratory for Communication Networks
KTH Royal Institute of Technology, Stockholm, Sweden
e-mail: etin@kth.se, stadler@kth.se

Abstract

We present a decentralized design for a self-organizing server cluster that supports a single service with response time guarantees. We advocate a decentralized approach, because it enables scalability and fault-tolerance, and has a lower configuration complexity than a centralized solution. Three distributed mechanisms form the key elements of the design. Topology construction maintains a dynamic overlay of cluster nodes. Request routing directs service requests towards servers with available capacity. Membership control allocates/releases servers to/from the cluster, in response to changes in the external load. Management policies can be changed at run-time using an epidemic protocol, and a distributed aggregation technique is introduced to enable external monitoring. We demonstrate through simulations that our system operates efficiently by comparing it to an ideal centralized system. In addition, we show the capabilities of the system to self-organize and adapt. The system effectively adapts to changing load and failures, even massive ones. Finally, we demonstrate the reaction of the system to a change of a global control parameter and its capability to monitor, at run time, global performance parameters including the average response time and the system size. We found that the interaction of the various mechanisms in the system leads to desirable global properties. For instance, for a fixed connectivity (which is the number of neighbors of a node in the overlay), the average experienced delay in the cluster is independent of the external load. In addition,

This technical report is an extension of a paper that has been published in IM-2005 and won the best student paper award.
increasing the connectivity increases the average delay but decreases the system size for a given load. Therefore, we use the connectivity as a management parameter that permits controlling the tradeoff between a smaller system size (more efficient operation) and a smaller experienced delay (better service for the customer).

6.1 Introduction

Internet service providers run a variety of applications on large server clusters. For such services, customers increasingly demand QoS guarantees, such as limits on the response time for service requests. In this context, a key problem is designing self-organizing server clusters that operate efficiently under QoS objectives. Self-organization relates to the ability of the cluster’s control system to effectively adapt to changes in load and to failures, i.e., to the capability of the system for self-optimization and self-healing. At the same time, the system must be externally controllable and observable and thus allow for run-time changes in management policies.

In this paper, we present and evaluate a decentralized approach to this problem. We choose a decentralized design for two reasons. First, recent research in areas including peer-to-peer systems and distributed management has demonstrated the benefits of decentralized over centralized designs: a decentralized design can reduce the configuration complexity of a system and increase its scalability and fault-tolerance. Second, we believe the study of self-organizing server clusters to be a first step towards developing fundamental concepts for autonomic systems in large-scale dynamic environments, which are decentralized by nature.

Advanced architectures for cluster-based services ([27], [24], [88], [33], [34], [28]) allow for service differentiation, server overload control and high utilization of resources. In addition, they are controllable in the sense that the system administrator can change, at runtime, the policies governing service differentiation and resource usage. These systems, however, do not have built-in architectural support for automatic reconfiguration in case of failures or addition/removal of system components. In addition, they rely on centralized functions, which limit their ability to scale and to tolerate faults.

Such limitations have been addressed in the context of peer-to-peer systems. However, even though peer-to-peer networks efficiently run best-effort services ([89], [49]), no results are available on how to achieve service guarantees and service differentiation using peer-to-peer middleware. In addition, as peer-to-peer systems enable, by design, a maximum degree of autonomy, they lack management support for operational monitoring and control, which is paramount in a commercial environment.

Our research aims at engineering the control infrastructure of a server cluster that provides the same functionality as the service architectures mentioned above (service differentiation, QoS objectives, efficient operation and controllability), while, at the same time, assumes key properties of peer-to-peer systems (scalability and adaptability).
6.1. INTRODUCTION

Figure 6.1: We present a design for large-scale, self-configuring node clusters. The design includes the nodes in data centers, the entry points and a management station.

Fig. 6.1 positions our work in the context of web service architectures. A layer 4 switch distributes the service requests, which originate from clients on the Internet, to the nodes of a server cluster. Some services may include operations that access or update a database through an internal IP network.

This paper focuses on the design of the server cluster in this three-tiered architecture. We assume the cluster to provide "computational" services, such as remote computations, or services that process online requests and generate dynamic content (online tax filing, e-commerce, etc.). For such services, a server must allocate some of its resources for a specific time in order to process a request. To reduce complexity, service requests with the same resource requirements are grouped into a single service class. All requests of a given service class have the same QoS constraint.

Three distributed mechanisms form the core of our design. Topology construction, based on an epidemic protocol, maintains a dynamic overlay of cluster nodes. Request routing directs service requests towards servers with available resources. Membership control allocates/releases servers to/from the cluster, in response to changes in the external load. We disseminate updates of control parameters and monitor a set of global system parameters using an epidemic protocol.

We have evaluated our design through extensive simulations under a variety of load patterns and failures. The metrics used for the evaluation include the rate of
rejected requests, the average delay per processed request, and the system size, i.e., the number of active servers. The simulations show that the system adapts fast to changes, and operates efficiently compared to an ideal centralized system.

We have discovered through simulation that the connectivity parameter $c$, which controls the number of neighbors of a node in the overlay network, has interesting properties. First, increasing the value of $c$ decreases the system size (i.e., the number of active servers), while the average response time per request increases. Second, for a given $c$, the average response time per request is independent of the system load. These properties make $c$ an effective control parameter in our design, since it allows controlling the experienced QoS per request (in addition to the QoS guarantee, which is an upper bound!). This parameter thus allows a cluster manager to control the tradeoff between a smaller system size (more efficient operation) and a smaller average response time (better service for the customer).

With this paper, we make the following contributions. We present a decentralized design of the control system for a self-organizing server cluster. The cluster offers a single service and guarantees a maximum response time to service requests. The system operates efficiently and dynamically adapts to changes in the external load pattern and to server failures. In addition, global control parameters can be changed at run-time and performance parameters can be continuously monitored from a management station.

This paper includes a significant extension of earlier results reported in [90].
6.2. SYSTEM DESIGN

Apart from smaller modifications to the basic design, we evaluate the system's response to failures, include results for servers with high processing capacities, and, further, provide and evaluate concepts for external control and monitoring.

The rest of this paper is structured as follows: Section 6.2 describes the system design. Section 6.3 presents an evaluation of the system through simulation and discusses the simulation results. Section 6.4 presents and evaluates schemes that allow for external control and observation of the system's behavior. Section 6.5 reviews related work. Finally, Section 6.6 contains additional comments and outlines future work.

6.2 System Design

System Model

We consider a system that provides a single service. It serves a stream of requests with identical resource requirements and guarantees a (single) maximum response time for all requests it processes. Following Fig. 6.1, we focus on a decentralized design for a cluster of identical servers. Each server is either in active mode, in which it is exchanging state information with other active servers, or in standby mode, in which it does not maintain any internal state related to the service. Only active servers process service requests. A standby server becomes active when it receives a join request from an active server. An active server switches to standby mode when its utilization becomes too low.

Service requests enter the cluster through a layer 4-7 switch, which assigns them to active servers in a uniform random fashion.

Active servers with high utilization redirect the assigned requests to other active servers. The processing of a request must complete within the maximum response time. Since each redirection induces an additional delay, too many redirections result in the violation of the maximum response time constraint, and the system rejects the request.

Let \( t_{\text{net}} \) be the upper bound for the roundtrip networking delay between client and cluster. The response time experienced by the client is then below \( t_{\text{net}} + t_{\text{cluster}} \), where \( t_{\text{cluster}} \) is the time the request spends inside the cluster. In the remainder of the paper, the term response time refers to \( t_{\text{cluster}} \).

Server Functionality

Each server runs two local mechanisms - the application service that processes the requests and the local admission controller that enforces the QoS constraints and decides, for each incoming request, whether the node should schedule and process it locally, or whether it should redirect it. Every server also runs an instance of the three decentralized mechanisms that control the behavior of the system: overlay construction, request routing and membership control. We will describe these three mechanisms in detail later in this section.
CHAPTER 6. EXTERNALLY CONTROLLABLE, SELF-ORGANIZING SERVER CLUSTERS

Table 6.1: The neighborhood table.

<table>
<thead>
<tr>
<th>Server ID</th>
<th>Timestamp</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>id₁</td>
<td>ts₁</td>
<td>u₁</td>
</tr>
<tr>
<td>id₂</td>
<td>ts₂</td>
<td>u₂</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>idₙ</td>
<td>tsₙ</td>
<td>uₙ</td>
</tr>
</tbody>
</table>

Figure 6.3: State synchronization on neighborhood tables.

Server State

Each node maintains a local neighborhood table, which contains an entry for the node itself and an entry for each of its neighbors in the overlay network (see Table 6.1). Each entry of the neighborhood table has a node identifier, a timestamp with the local time of its latest update, and a data field with additional information about the node to which the entry refers. In our current design, the data field contains the node’s utilization.

The overlay construction mechanism periodically rebuilds the neighborhood table on each node and, during such a run, likely changes its set of neighbors. After a run of the overlay construction mechanism, a node synchronizes its state with that of its neighbors by requesting their current utilizations and updating the data
fields in its neighborhood table accordingly (see Fig. 6.3). In Section 6.4, this synchronization scheme is extended to include control and monitoring parameters for management.

Decentralized Control Mechanisms

The Overlay Construction Mechanism organizes all active nodes in an overlay network in which each node has an equal number of logical neighbors. We call the number of neighbors of a node the connectivity $c$ of the overlay network. The overlay construction mechanism is based on Newscast [59], an epidemic protocol, and works as follows. Periodically, each node exchanges its neighborhood table with a randomly chosen neighbor. After the exchange, a node rebuilds its neighborhood table by selecting the $c$ neighbor entries with the most recent timestamps from the union of the two original neighborhood tables. This way, both nodes end up with identical tables. This protocol is simple, yet very robust, as it adapts rapidly to node additions/removals and node failures [59]. Note that the use of timestamps gradually erases old state information from the neighborhood tables.

The overlay construction mechanism has two control parameters that influence the global behavior of the cluster: the connectivity parameter $c$ and the time $\Delta t$ between two successive runs of the mechanism on a node.

The Request Routing Mechanism directs requests towards available resources. Routing is activated when the admission controller rejects a request due to insufficient local resources. The node checks its neighborhood table for neighbors with utilization below a threshold $u_{\text{max}}$. If there are such neighbors, the node picks one at random and forwards the request. Otherwise, it picks at random any neighbor and sends the request to that node. In order to avoid routing loops, each request keeps in its header the addresses of the servers it has visited.

The routing mechanism occasionally rejects a request from the cluster, which happens in the following cases. If a node cannot process a service request that has already visited all of its neighbors, then the request is rejected. In addition, a request that has been redirected many times will be rejected, once the maximum response time can no longer be met.

The Membership Control Mechanism adjusts the system size, i.e., the number of active servers in the system, in function of the utilization information in the neighborhood tables. Every time its neighborhood table changes, a node invokes the membership control mechanism. The mechanism compares the node’s utilization to a threshold $u_{\text{min}}$. If it is below $u_{\text{min}}$, the node switches to standby and leaves the cluster. Otherwise, the node remains active. Furthermore, if a node’s utilization and the utilizations of its neighbors (as stored in the neighborhood table) are above a second threshold $u_{\text{max}}$, then it requests a standby node to become active. The membership control mechanism regulates the system in such a way that the utilization of all active nodes tends to fall within the interval $[u_{\text{min}}, u_{\text{max}}]$. 
CHAPTER 6. EXTERNALLY CONTROLLABLE, SELF-ORGANIZING SERVER CLUSTERS

6.3 System Evaluation through Simulation

We have implemented our design in Java and studied its behavior through simulation. The simulation scenarios described in this section test the capability of the cluster to reconfigure dynamically in response to load fluctuations and node failures, while supporting QoS constraints.

Performance Metrics

We use the following metrics to evaluate the system behavior: the request rejection rate, the average response time per processed request, and the system size. Under constant external conditions, we also measure the distribution of the response times for all the processed requests. The rejection rate and the average response time measure the experienced quality of service. (The cluster guarantees by design the maximum response time.) The system size measures how efficiently the system provides the service.

Ideal System

To assess the relative efficiency of our design, we compare the measurements taken from our system to those of an ideal system. An ideal system does not drop requests, processes all incoming requests without queuing delay, and instantly adjusts its size to the minimum possible value in response to changes in the external load. We cannot engineer such a system, of course, but it serves as a lower bound for the performance metrics of our cluster.

Connectivity Parameter

We have run every simulation scenario for different values of the connectivity parameter $c (c = 0, 5, 20, 50)$. The case $c = 0$ corresponds to a system in which the overlay construction and request routing mechanisms are not activated and nodes do not redirect requests.

If the connectivity equals the system size, then each node knows about every other node, and the overlay is a full mesh. If the connectivity is larger than the system size, the nodes never fully populate their neighborhood tables, but still know about every other node in the system. In such a case, we say that the system has an effective connectivity equal to its size.

Simulating the System Design

We use javaSimulation, a Java package that supports process-based discrete event simulation [17]. Three types of simulation processes run on top of javaSimulation: the cluster, the request generator, and the server. The cluster starts the simulation and creates instances of the request generator and the servers (one process per server). The request generator simulates a layer 4 switch that receives service
6.3. SYSTEM EVALUATION THROUGH SIMULATION

Figure 6.4: Results for the steady load scenario.

requests from external clients, modeled through a Poisson process, and assigns these requests to the servers following a uniform random distribution. Each server runs local mechanisms (admission control and request processing) and distributed mechanisms (overlay construction, request routing, and membership control) as described in Section 6.2.

Every simulation starts with a pool of 300 active servers with empty neighborhood tables. The system warms up for 100 sec. During this time, the nodes fill their neighborhood tables and construct the overlay. The overlay construction mechanism runs every 5 sec on each node.

Every request has an execution time of 1 sec on a server, and the maximum response time per request is 2 sec. Consequently, the maximum time a request can spend in the cluster before its processing must start is 1 sec. This time includes the delays for locally handling the request (admission control, scheduling, routing)
and the communication delay between overlay neighbors. We use 0.167 sec per hop for handling and communication, which limits to five the path length of a request in the cluster.

Server Capacity

The maximum number of requests per sec that a server can process depends on (a) its resources, such as CPU, memory, disk space, and (b) the type of service the system provides. Therefore, we are using the term capacity in a relative sense for our evaluation. A high-end server running an application service with strong demand on resources can have the same capacity as a mid-range server executing a less-demanding application. In the scenarios for evaluating our design we consider two types of servers: low-capacity servers, which process up to 4 requests/sec, and high-capacity servers, which process up to 40 requests/sec.

Evaluating the Basic Design for Low-capacity Servers

In this subsection, we evaluate the ability of the system to adjust its size in response to fluctuations in the external load. We use the following three simulation scenarios: steady load, rising load, and dropping load.

The steady load scenario evaluates the performance of the system under a constant average load of $\lambda_0=200$ requests/sec.

The rising load scenario starts with a request arrival rate $\lambda_1=100$ requests/sec for the first 300 sec. At that time, the request arrival rate rises instantly to $\lambda_2=333$ requests/sec, following a step function. This scenario examines the ability of the system to add new resources in order to meet a steep increase in external demand for a service.

The dropping load scenario reverses the rising load scenario. It starts with a request arrival rate of $\lambda_2=333$ requests/sec for the first 300 sec. At that time, the request arrival rate drops instantly to $\lambda_1=100$ requests/sec, following a step function. This scenario evaluates the ability of the system to release resources when the external demand for a service suddenly drops.

Influence of the External Load on System Behavior

The steady load scenario shows that the system is stable under a constant average load.

The rising and dropping load scenarios show that the system stabilizes after abrupt changes in external load within some 15 sec, which corresponds to 3 iterations of the overlay construction mechanism. The system then exhibits steady-state behavior.

In the rising load scenario, the sudden rise in demand for the service causes a spike in the rate of rejected requests and in the experienced average delay. The system tends to "overshoot" while adjusting its size. We observe that a larger
6.3. SYSTEM EVALUATION THROUGH SIMULATION

![Graphs](image)

(a) average response time  (b) average response time

(c) rejection rate  (d) rejection rate

(e) system size  (f) system size

Figure 6.5: Left: results for the rising load scenario. Right: results for the dropping load scenario.
connectivity parameter results in a longer settling time. Because of the overshoot in system size, a temporary increase in service quality occurs: the average response time and the rejection rate are lower during the settling period.

In the dropping load scenario, the average response time decreases significantly after the drop in demand for the service and increases later to previous levels.

**Influence of the Connectivity Parameter on System Behavior**

The measurements from the above simulation scenarios suggest that the system has a remarkable global property: for a fixed value of the connectivity parameter, the average response time is independent of the external load. As Figs. 6.4, 6.5 show, a given connectivity results in the same average response time (1.25 sec for \( c = 5 \), 1.35 sec for \( c = 20 \), and 1.45 sec for \( c = 50 \)) for three different load patterns with the arrival rates \( \lambda_0 \), \( \lambda_1 \) and \( \lambda_2 \). The lower average value of 1.4 sec obtained for \( c = 50 \) and the arrival rate \( \lambda_1 \) is explained by the fact that the system size is smaller than the value of the connectivity parameter. In this case, the system operates with an effective connectivity of 32. The simulation scenarios also suggest that the value of the connectivity parameter \( c \) directly affects the system size. A larger value for \( c \) results in a smaller system size.

In order to confirm these observations we have conducted additional simulations with constant average loads. In these simulation runs, the arrival rate \( \lambda \) takes values between 100 and 1600 requests/sec, and the connectivity takes values between 0 and 50. For each pair \((\lambda, c)\), we have computed the average size of the system and the average response time per processed request over 5000 sec. The results are presented in Fig. 6.6.
6.3. SYSTEM EVALUATION THROUGH SIMULATION

The graph in Fig. 6.6 shows that, for a fixed request arrival rate, the size of the system decreases as the connectivity increases. It also shows that, for a fixed connectivity, the average response time stays within 0.25% of a fixed value, regardless of the arrival rate.

The exceptions to this rule are the measurement points in the upper left corner, corresponding to $\lambda = 100$ and $c = 30, 40, 50$. For these points, the size of the system is smaller than the connectivity, and the system operates with an effective connectivity equal to its size.

Based on the graph in Fig. 6.6, we conclude that the connectivity can be used as a management parameter. When the manager increases the value of the connectivity, the system becomes more efficient, but the quality of service experienced by the clients decreases. Decreasing the connectivity has the opposite effect. We will discuss the realization of such a control capability in Section 6.4.

A further observation related to the connectivity parameter can be made using Fig. 6.4(b). The distribution of the response times indicates that the system redirects more requests for larger values of the connectivity. More precisely, the percentage of redirected requests is 1.6% for $c = 5$, 6.3% for $c = 20$, and 12.8% for $c = 50$. The distribution of the response times of the processed requests has five peaks for $c = 50$. The number of peaks in the graph is equal to the maximum path length of a request. The first peak appears at 1.167 sec, which corresponds to the case where the first server immediately processes the request. (1 sec needed for processing the request and 0.167 sec for admission control and scheduling.) Similarly, the following peaks, at the response-time values of $(1 + n * 0.167)$ sec, with $n = 2, 3, 4, 5$, correspond to the cases where a request is redirected $n$ times, before being scheduled for immediate processing on the $n^{th}$ server. The distribution for $c = 5$ has 2 peaks, as only 0.01% of the requests are redirected more than 2 times.

**System Recovery after Failures**

The failure scenario described in this subsection evaluates the ability of the system to quickly detect node failures and recover through reconfiguration. We simulate a constant average load of $\lambda_0 = 200$ requests/sec for the scenario. After the first 300 sec, a subset of the active servers fails. A failed server loses its state and does not participate in the service anymore. Consequently, the remaining active servers experience a surge in utilization and request standby servers to join the cluster. For the scenario, we assume that the pool of standby servers is large enough to replace all failed servers.

Fig. 6.7 shows the results for two runs of the failure scenario, in which either 30% or 70% of the active servers fail at a specific time. In both runs, the system recovers from failures in about 15 sec, which corresponds to 3 iterations of the overlay construction mechanism.

Server failures results in a spike of in the rate of rejected requests and in average delay. We observe that the system behavior in the failure scenario is similar to that in the rising load scenario (Fig. 6.5). We explain this by the fact that the
Figure 6.7: Results for the failure scenario. Left: 30% of the nodes fail. Right: 70% of the nodes fail.
server failures leave the system with a size that is too small to handle the incoming requests, and it, therefore, reacts in a similar way as when the external load suddenly rises. As in the rising load scenario, the system overshoots when re-adjusting its size after the failures. The size of the overshoot, as well as the settling time, increases with a larger connectivity and a larger percentage of failed servers.

### 6.4 External Control and Monitoring of the System

Although the system presented in Section 6.2 is self-organizing, external control and monitoring are highly desirable to change dynamically global control parame-

---

**Table 6.2: Local Control Table (LCT)**

<table>
<thead>
<tr>
<th>Control Parameter</th>
<th>Last Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectivity</td>
<td>( t_{connectivity} )</td>
</tr>
<tr>
<td>( \Delta t_{overlay} )</td>
<td>( t_{\Delta t} )</td>
</tr>
<tr>
<td>min_util</td>
<td>( t_{min_util} )</td>
</tr>
<tr>
<td>max_util</td>
<td>( t_{max_util} )</td>
</tr>
<tr>
<td>max_resp_time</td>
<td>( t_{max_resp_time} )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Table 6.3: Local State Table (LST)**

<table>
<thead>
<tr>
<th>Monitoring Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>utilization</td>
</tr>
<tr>
<td>avg_resp_time</td>
</tr>
<tr>
<td>rejection_rate</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

**Table 6.4: Global State Table (GST)**

<table>
<thead>
<tr>
<th>Monitoring Parameter</th>
<th>Last Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg_resp_time</td>
<td>( t_{avg_resp_time} )</td>
</tr>
<tr>
<td>rejection_rate</td>
<td>( t_{rej_rate} )</td>
</tr>
<tr>
<td>system_size</td>
<td>( t_{system_size} )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
CHAPTER 6. EXTERNALLY CONTROLLABLE, SELF-ORGANIZING SERVER CLUSTERS

ters and to monitor the performance of the system. We present a decentralized design for managing the system from an external management station. The design is based on the mechanisms introduced in Section 6.2. It includes data structures for holding management parameters on the servers and schemes for disseminating control parameters and estimating state variables. As a basic principle, the management station can contact any active server in the cluster for the purpose of changing a management parameter in the system or reading an estimate of a global performance metric.

System Control

Our design enables the cluster administrator to change the behavior of the system at runtime by sending a command to an active server, which results in modifying the value of a control parameter on every active server of the cluster. As demonstrated by the measurements compiled in Fig. 6.6, the connectivity $c$ can be used as a management parameter that allows a cluster administrator to control the tradeoff between a smaller system size and a smaller average response time.

For management purposes, each (active) server maintains a local control table. Each entry in this table contains information about a management control parameter: its name, its value, and the timestamp of its last update. Table 6.2 shows this table with five entries: the connectivity of the overlay, the time scale $\Delta t$ of the overlay construction mechanism, the maximum response time per request, and the thresholds for minimum and maximum server utilization. These parameters control the behavior of mechanisms on the server, including topology construction, routing and membership control.

A control operation changes a specific parameter in the local control table of every active server. The management station issues a management command with a unique time stamp to an active server, which updates the corresponding entry of the server’s local table.

We have investigated two different schemes that disseminate new values for control parameters in the cluster. The first scheme extends the overlay construction mechanism described in Section 6.2. When a node exchanges and synchronizes its local neighborhood table with that of a neighbor, the local control table is included in this operation. This means that a server synchronizes its local control table with that of a neighbor by comparing the timestamps for each parameter in both tables. If the timestamp in the neighbor’s table is more recent, the entry in the neighbor’s table overwrites the local entry. This way, both nodes end up with identical tables. As the overlay construction mechanism is based on Newcast [59], we call this scheme the Newcast scheme.

The second scheme, which we call Neighborcast, extends the scheme shown in Fig 6.3. A server contacts its neighbors, one by one, to read their current utilization values. While a server contacts a neighbor, both nodes synchronize their local control tables as in the Newcast scheme. The exchange of control tables is illustrated in Fig 6.8. Since such a round of synchronizations involves all the
neighbors of a node, we expect this scheme to disseminate updates faster than the Newscast scheme.

Note that neither scheme adds additional messages to the operation of the system, but relies on the messages for updating the local neighborhood tables described in Section 6.2. Both schemes increase the size of these messages though.

We compare the two dissemination schemes in a simulation scenario with a constant average load of $\lambda=200$ requests/sec and an average system size of 82 nodes. After 250 sec, the manager changes on a server the value of the connectivity from 5 to 20. As a result, the system reconfigures and the average system size drops to some 65 nodes. Fig. 6.9 shows how the new value spreads to all active nodes. (Note that the set of active nodes changes dynamically, even in steady state.) As expected, Neighborcast converges faster than the Newscast scheme. After some 10 sec, which includes 2 rounds of synchronizations, at least 99% of the nodes have received the new value. It takes between 35 and 40 sec for the Newscast scheme to achieve the same coverage.

We evaluate the controllability of the system using the same simulation scenario as above. It simulates a case where the cluster manager, using the information in Fig. 6.6, decides to change the system’s operating point with the goal of decreasing its size by some 20%, while allowing the average response time to increase by some 0.1 sec. To achieve this, the manager changes the connectivity from 5 to 20. Neighborcast is used to disseminate the change in connectivity.
CHAPTER 6. EXTERNALLY CONTROLLABLE, SELF-ORGANIZING SERVER CLUSTERS

![Graph showing % of nodes updated over simulation time for Newscast and Neighborcast](image)

**Figure 6.9:** Propagation to the changes of a control parameter.

(a) average response time  
(b) system size

![Graphs showing average response time and system size over simulation time](image)

**Figure 6.10:** Reaction of the system to changes in the connectivity parameter.

Fig. 6.10 gives the measurements from this scenario. The system reaches a steady state after 100 sec. At 250 sec, the connectivity parameter on an active node is changed, and the system enters a transitional phase, during which the new value of the connectivity parameter spreads and the nodes adjust their neighbor and management tables. (Fig. 6.9 provides the measurements from the dissemination.) The system stabilizes around the new operating point after some 50 sec, which corresponds to 10 runs of the membership control mechanism. At 425 sec, the value of the connectivity parameter is changed back from 20 to 5 on an active server, and the system returns to the previous state after a settling time of about
40 sec. Fig. 6.10 does not include the chart for the rejection rate, since virtually no requests were rejected during this scenario. (Out of 3 million generated requests, 4 were rejected.)

System Monitoring

We extend our design with a decentralized monitoring capability that allows a management station to read current estimates of global performance parameters, such as the average response time, the rejection rate, and the system size. The core problem to solve is that of estimating, from local node states, the global state of a system whose set of nodes and (overlay) topology continues to change. Many recent research results from topics, such as fault-tolerant spanning trees on dynamic graphs or decentralized aggregation using epidemic protocols, are potentially applicable to decentralized monitoring, and we expect rapid progress in this area in the near future. We selected two specific schemes, adapted them to our design, and evaluated them in simulation scenarios.

A sampling scheme: A straightforward way of estimating a global parameter that is an average of local state variables is by sampling a small set of servers. Our experience has shown that, for parameters including average response time and rejection rate, a simple sampling scheme can provide quite accurate estimates. We explain this by the fact that the system balances the load well among the servers. In the specific scheme we implemented, a node, when contacted by a management station about an estimate, retrieves a local variable, as well as the local variable of each of its neighbors, computes the average of all the values, and returns the result.

An epidemic scheme: Epidemic protocols have been recently proposed to compute, in a decentralized way, minima, maxima and average values of local variables across a set of interconnected nodes (see, e.g., [91] [92]). Advantages of an epidemic scheme over sampling are that the former can produce accurate estimates independent of the variance of the local variables and that the system size can be computed. A drawback of an epidemic scheme is that it often needs a large number of rounds for a local variable to converge to within a given interval of the real value [91]. We believe that an epidemic scheme is an interesting candidate to estimate certain global variables for a system in steady state, but it remains to be investigated whether such a scheme is suitable in dynamic environments such as ours.

Epidemic protocols can be used to compute aggregates as follows [91]. Periodically, each node selects a random neighbor and exchanges with it the value of a specific local variable. Both nodes aggregate the two values, using the average, minimum, or maximum function, and store the results in the local variable. It can be shown that, over time, the local variable in each node converges towards the global aggregate, i.e. the aggregate of the values of all local variables at the beginning of the computation [91]. The system size can be evaluated starting with an initial peak distribution: one node holds a value of 1, and all other nodes hold
values of 0. Applying the average as the aggregation function, all local variables converge towards the inverse of the system size.

We have extended our design to include an epidemic scheme for estimating performance parameters. For this purpose, each server maintains two tables (Tables 6.3, 6.4): a local state table, which contains statistics from its own operation, and a global state table, which contains aggregates of local statistics from different nodes and thus estimates of global state variables.

In our design, the global state tables are exchanged between neighbors using Neighborcast, i.e., in the same ways as the local control tables are exchanged. A round of exchanges occurs after each run of the topology construction mechanism. Having retrieved the global state table from a neighbor, a node re-computes the values of its global state table using the average as aggregate function.

When estimating the system size, we need to take into account that servers may leave or join the cluster during the estimation process. A server that joins the cluster, i.e., changes from standby to active, has its size variable initialized to 0. A server that leaves the cluster sends the value of its system size variable to its neighbors before switching to standby. After that, the neighbors increase the values of their variables, so that the sum of the system size variables across all active servers remains 1. Note that this procedure does not provide a good estimation of the system size in case of server failures in which the local states are lost. In such a case, the process for estimating the system size must be restarted.

We evaluate both monitoring schemes using the above simulation scenario with a constant average load of $\lambda=200$ requests/sec. After 350 sec, the manager changes the connectivity from 5 to 20. Neighborcast distributes the change in connectivity.

Fig. 6.11 shows the results for monitoring the average response times and the system size. The curves marked as "real" give the actual values of the parameters as computed by polling each server, whereas the curves marked as "estimated" show the values obtained by our monitoring schemes. The average response time is estimated using the sampling scheme, and 98% of the measurements lie within 5% of the real values of the parameter. The accuracy of this scheme can be further increased by sampling more nodes or by applying results from estimation theory.

The system size is estimated using the epidemic scheme. The analysis shows that 90% of the values for the estimated size lie within 5% of the real size. The estimates are more accurate for larger values of the connectivity.

6.5 Related Work

Various aspects of our design relate to platforms for web services with QoS objectives, peer-to-peer systems, applications of epidemic protocols, and activities within the grid computing community.
6.5. RELATED WORK

Centralized Management of Web Services with QoS Guarantees

In [27] and [28], the authors propose centralized resource management schemes for balancing service quality and resource usage. Both systems attempt to maximize a utility function in the face of fluctuating loads.

In [27], a performance management system for cluster-based web services is presented. The system dynamically allocates resources to competing services, balances the load across servers, and protects servers against overload. The system described in [28] adaptively provisions resources in a hosting center to ensure efficient use of power and server resources. The system attempts to allocate dynamically to each service the minimal resources needed for acceptable service quality, leaving surplus resources available to deploy elsewhere.

As in our design, both approaches described in [27] and [28] map service requests into service classes, whereby all requests in a service class have the same QoS objective.

The cluster architecture in [27] contains several types of components that share monitoring and control information via a publish/subscribe network. Servers and gateways continuously gather statistics about incoming requests and send them periodically to the Global Resource Manager (GRM). GRM runs a linear optimization algorithm that takes as input the statistics from the gateways and servers, the performance objectives, the cluster utility function, and the resource configuration. GRM computes two parameters: the maximum number of concurrent requests that server $s$ executes on behalf of the gateway $g$ and the minimum number of class $c$ requests that every server executes on the behalf of each gateway. GRM then forwards the new parameter values to the gateways, which apply them until they receive a new update.
Similarly, the Muse resource management scheme ([28]) contains several types of components: servers, programmable network switches, and the executor. Servers gather statistics about incoming requests and process assigned requests. The programmable switches redirect requests towards servers following a specific pattern. Finally, the executor analyzes bids for services from customers and service statistics from servers and periodically computes an optimal resource allocation policy.

Two main characteristics distinguish our design from these two approaches: our design is decentralized, and all our cluster components are of the same type. We believe that our approach leads to a lower system complexity and thus the task of configuring the system becomes simpler. In addition, it eliminates the single point of failure, namely, GRM in [27] and the executor in [28].

**Structured Peer-to-Peer Systems**

As mentioned above, our design shares several principles with peer-to-peer systems. As part of this work, we have studied the possibility of developing a decentralized architecture for server clusters with QoS objectives on top of a structured peer-to-peer system. We concluded that such an approach would likely lead to a system that is more complex and less efficient than the one presented in this paper, and we explain here briefly why. (To keep the term short, we use peer-to-peer system instead of structured peer-to-peer system.)

Peer-to-peer systems are application-layer overlays built on top of the Internet infrastructure. They generally use distributed hash tables (DHTs) to identify nodes and objects, which are assigned to nodes. A hash function maps strings that refer objects to a one-dimensional identifier space, usually \([0, 2^{128} - 1]\). The primary service of a peer-to-peer system is to route a request with an object identifier to a node that is responsible for that object. Routing is based on the object’s identifier and most systems perform routing within \(O(\log n)\) hops, where \(n\) denotes the system size. Routing information is maintained in form of a distributed indexing topology, such as a circle or a hypercube, which defines the topology of the overlay network.

If one wanted to use a peer-to-peer layer as part of the design of a server cluster, one would assign an identifier to each incoming request and would then let the peer-to-peer system route the request to the node responsible for that identifier. The node would then process the request. In order for the server cluster to support efficiently QoS objectives, some form of resource control or load balancing mechanism would be needed in the peer-to-peer layer.

Introducing load-balancing capabilities in DHT-based systems is a topic of ongoing research ([52], [93], [94]). An interesting result is that uniform hashing by itself does not achieve effective load balancing. In [52], the authors show that, in a network with \(n\) nodes, where each node covers on average a fraction of \(1/n\) of the identifier space, with high probability, at least one node will cover a fraction of \(O(\log n / n)\) of the identifier space. Therefore, uniform hashing results in an \(O(\log n)\) imbalance in the number of objects assigned to a node. Recently proposed solutions
6.5. RELATED WORK

to the problem of load balancing in DHT systems include load stealing schemes [93] or "the power of two choices" [52].

In order to implement an efficient resource allocation policy that dynamically adapts to external load conditions, the identifier space in a peer-to-peer system needs to be re-partitioned and the partitions reallocated on a continuous basis. This means that the indexing topology, a global distributed state, needs to be updated continuously to enable the routing of requests to nodes.

When comparing request routing based on DHTs with the routing mechanism in our design, we concluded that maintaining a global indexing topology is significantly more complex than maintaining the local neighborhood tables in our design.

In addition, peer-to-peer systems have properties that are not needed for our purposes. For instance, a peer-to-peer system routes a request for an object to a particular server - the one that is responsible for that object. (This property is useful to implement information systems on peer-to-peer middleware.) In our design, a request can be directed to any server with available capacity, which simplifies the routing problem in the sense that there is no need to maintain a global distributed state.

Epidemic Protocols

Epidemic algorithms disseminate information in large-scale settings in a robust and scalable way. The use of epidemic algorithms has been studied for applications such as data aggregation [95], resource discovery and monitoring [13], database replication [56], [57], populating routing tables in peer-to-peer systems [96], and handling web hotspots [58].

We apply an epidemic algorithm, Newscast [59], to locate available resources in a system that is subject to rapid changes. In our design, the topology construction mechanism uses Newscast to construct and maintain the overlay, through which requests are being routed. We further use Newscast as a basis for disseminating control parameters and as part of an aggregation scheme to estimate global state variables.

Grid Computing

As in our system, a grid locates available resources in a network of nodes for processing requests (or tasks). Resource management and task scheduling in distributed environments is a fundamental topic of grid computing. The goal often is to maximize a given workload subject to a set of QoS constraints. Many activities ([60], [61], [62]) focus on statistical prediction models for resource availability. These models are used as part of centralized scheduling architectures, in which a single scheduler dispatches tasks to a set of available machines. Other research ([64]) creates peer-to-peer desktop grids that, however, do not work under QoS constraints.
CHAPTER 6. EXTERNALLY CONTROLLABLE, SELF-ORGANIZING
SERVER CLUSTERS

There is comparatively little research in the area of management of the grid
computing. The largest grid testbed, Grid3 [97] has a centralized database that
gathers information about system activity from the participating sites.

6.6 Discussion and Future Work

In this paper, we presented key elements of a design for an externally controllable,
self-organizing server cluster that supports a single service with response time guar-
antees. The design is simple, as it relies on three modular distributed mechanisms
that can be written in a few lines of pseudo code.

While the design itself is simple, the global behavior of the system, which results
from the interactions among these mechanisms in response to external events, is
difficult to understand. We have used extensive simulations to study the system
behavior and evaluate its performance characteristics.

We have been surprised by how efficient the system performs compared to an
ideal centralized server. In the scenarios studied, the connectivity can be increased
to a level at which the system size falls within 20% of the theoretical minimum of
an ideal centralized server.

In addition, the system adjusts fast to changes in load patterns and to faults.
When running topology construction every 5 sec, the system size stabilizes within
15-20 sec after sudden changes in the load and after failures (even massive failures
in which 70% of the active servers fail).

We discovered that the connectivity parameter \( c \) allows controlling the average
experienced delay per service request. Remarkably, this delay has shown to be
independent of the external load in all scenarios studied. Since the system size
depends on \( c \), the cluster administrator can use connectivity as a management
parameter in order to control the tradeoff between a small system size and a small
experienced delay for the service.

We are aware that the design given in this paper is not complete. Problems not
addressed include specifying the mechanism used by the layer 4 switch to maintain
a list of the currently active servers and specifying the mechanism used by an active
server to identify a standby server. A possible approach for the layer 4 switch could
be that it has knowledge of only a subset of active servers and thus directs requests
only to those. Such a subset would be easier to maintain and could be realized,
e.g., by including the layer 4 switch as a node in the management overlay. This
approach would increase the number of request redirections in the cluster and, as
a consequence, the average response time.

In addition, we did not address in this paper the so-called server affinity problem.
Our current design handles all service requests independently of one another and
thus has no concept of a session. Many practical applications, though, require a
series of requests to be executed within a session. In such a case, one might want
to process all of these requests on a single server and keep the session state between
requests on the server.
The paper evaluates the effectiveness of our design with regard to efficient use of cluster resources for processing external requests and with regard to changes in the environment. Currently lacking is a thorough evaluation of the cluster control mechanisms (topology construction, request routing, membership control, dissemination of control parameters, distributed estimation of global state variables) with respect to message and processing overhead. While the design assures that the communication load for all these mechanisms is evenly distributed across the links of the management overlay and that the processing load is evenly distributed among the active servers, the actual values for these loads depend on several parameters, such as the time scale of topology reconstruction and the connectivity.

We believe that further research is needed into effective distributed monitoring in dynamic environments. We have shown in this paper that an epidemic scheme can provide "reasonable" estimates of global state variables, such as the current system size, and that sampling techniques can be applied to estimate the average response time. It needs to be investigated how the accuracy of such estimates can be controlled and how they can be extended to provide accurate results in failure scenarios.

Most server clusters today run more than one service or application. We are therefore extending our design to support multiple services in a cluster (or, more precise, multiple classes of service), each with its own QoS objectives. In such a scenario, several services compete for resources within the same server pool. Our goal is to find a decentralized mechanism that efficiently allocates resources to the various services, following cluster objectives that are applicable in real scenarios. While results on centralized mechanisms for this purpose have been reported, decentralized control of a multi-service server cluster remains a significant challenge.

Finally, we plan to implement the system on our lab testbed, which includes some 60 rack-mounted PCs, and use it to run selected web service applications.
Chapter 7

A Middleware Design for Large-scale Clusters Offering Multiple Services

Constantin Adam and Rolf Stadler
Laboratory for Communication Networks
KTH Royal Institute of Technology, Stockholm, Sweden
e-mail: ctin@kth.se, stadler@kth.se

Abstract

We present a decentralized design that dynamically allocates resources to multiple services inside a global server cluster. The design supports QoS objectives (maximum response time and minimum loss rate) for each service. A system administrator can modify policies that assign relative importance to services and, in this way, control the resource allocation process. Distinctive features of our design are the use of an epidemic protocol to disseminate state and control information, as well as the decentralized evaluation of utility functions to control resource partitioning among services. Simulation results show that the system operates both effectively and efficiently; it meets the QoS objectives and dynamically adapts to load changes and to failures. In case of overload, the service quality degrades gracefully, controlled by the cluster policies.

7.1 Introduction

The basic motivation of our work is to develop engineering principles for large-scale autonomous systems, where the individual components work together and coordinate their actions towards a common goal. We aim for a system design that is scalable, self-organizing and manageable. Scalability ensures that the processing
CHAPTER 7. A MIDDLEWARE DESIGN FOR LARGE-SCALE CLUSTERS
OFFERING MULTIPLE SERVICES

capacity of the system increases proportionally to the number of nodes, up to a very large size. Self-organization ensures that the system dynamically adapts to external events, such as changes in load patterns and component failures, by providing continuous service and making efficient use of its resources at all times. Finally, manageability ensures that the system is observable and controllable from an external management station.

We perform this research in the context of web services. Specifically, we consider services running on large clusters that are geographically dispersed and that contain multiple entry points. We assume these services to be computationally intensive, to be customized and to support QoS objectives, such as maximum response time for requests and minimum throughput. Examples of such services include remote computations and services with dynamic content, e.g., e-commerce, or online tax filing.

To reduce design complexity, service requests with the same resource requirements and QoS objectives are grouped into a single service class. Service classes are used to implement Service Level Agreements (SLAs), which specify a set of QoS objectives, as well as the consequences for exceeding, meeting, or violating these performance targets for a service. In this paper, we use the terms service and service class interchangeably.

Fig. 7.1 shows a generic deployment scenario for our design. It includes several data centers and has many entry points. We envision that the number of servers in such a system can reach hundreds of thousands. As a point of reference, Akamai's content delivery network currently contains some 20'000 servers [6].

Our approach is based on a decentralized design for managing and controlling such a cluster. We believe that a decentralized design can lead to efficient, scalable, self-configuring and robust systems. Our goal for these systems is to provide the same functionality as centralized service architectures ([27], [28], [24], [29]) - service differentiation, support for QoS objectives, efficient operation and controllability - while, at the same time, to exhibit key properties of peer-to-peer systems ([48], [49], [50], [51]) - minimal configuration complexity, scalability, adaptability and fault tolerance.

Three distributed mechanisms form the core of our design, as shown in Fig. 7.2. Topology construction, based on an epidemic protocol, organizes the cluster nodes into dynamic overlays, which are used to disseminate state and control information in a scalable and robust manner. Request routing directs service requests towards available resources. Service selection dynamically partitions the cluster resources between services in response to external events, by continuously maximizing utility functions.

We have evaluated our design through extensive simulations. In this paper, results from four specific scenarios are presented. We discuss the cluster behavior for scenarios, where the average load is constant, where a major load fluctuation occurs, where a large number of servers fail and where the system administrator changes the global cluster objective.
7.1. INTRODUCTION

With this paper, we make the following contributions. We present the decentralized design of the control system for a self-organizing server cluster. The cluster offers multiple services and efficiently allocates its resources between these services. Each service has two QoS objectives - the maximum response time and the maximum loss rate. Our design enables system administrators to change dynamically policies that assign importance to services and, this way, control the resource allocation process. In case of overload or failures, the QoS degrades gracefully, controlled by the relative importance of each service.

This paper significantly extends our earlier work presented in [90] and [98] in the following way. Our earlier work focuses on a design for a single service with QoS objectives. In this paper, we consider a cluster that provides support for multiple services, each with its own QoS objectives, and thus operates under a set of global objectives. The key additions in this work are the service selection mechanism and the use of utility functions to express the global system objectives.

The rest of this paper is structured as follows: Section 7.2 describes the system design. Section 7.3 describes the evaluation of the system behavior through simulation. Section 7.4 discusses the simulation results. Section 7.5 reviews related work. Finally, Section 7.6 contains additional observations and outlines future work.
CHAPTER 7. A MIDDLEWARE DESIGN FOR LARGE-SCALE CLUSTERS
OFFERING MULTIPLE SERVICES

7.2 System Design

System Model

Following Fig. 7.1, we consider a system that contains several data centers and many
entry points and refer to the collection of servers in these centers as the cluster.
Service requests enter the cluster through the entry points, which are typically layer
4-7 switches. An entry point advertises the cluster services to the outside world
and can perform additional functions, such as acting as a firewall. In our design,
it maps incoming requests into service classes and directs each request to a server
running that service. An entry point has knowledge of a set of servers assigned to
each service class. We will show later how the entry point maintains this knowledge.

In our design, the cluster attempts to process each request within a service-
specific response time; if this turns out not to be possible, it rejects the request.
Servers process requests in order of arrival. Upon receiving a request, a server
schedules it for execution, subject to the maximum response time objective. In case
of overload, the server redirects the request to another server that runs the same
service. Since each redirection induces an additional delay, too many redirections
results in the violation of the maximum response time objective, and the system
rejects the request.

In order to simplify the system model, we assume that: (a) all servers are
identical (i.e., have the same hardware/software resources and have the capability
to provide any of the services offered by the cluster), and (b) all requests require
the same amount of resources and time to execute. Consequently, each server can
process concurrently the same number of requests and has the same service rate (in
requests per second). In Section 7.4, we discuss the system behavior for requests
whose execution times follow an exponential distribution. It is straightforward to
extend our design for servers with various capacities or service types with various
resource requirements.

With this design, we attempt to manage a single resource: the CPU utilization.
Extending the model for several types of resources (CPU, memory, and network
bandwidth) is beyond the scope of this paper. The response time a client experi-
ences after sending a request is the sum of networking delays and processing delays.
Let \( t_{\text{net}} \) be the upper bound for the roundtrip networking delay between client and
cluster. The response time experienced by the client is then below \( t_{\text{net}} + t_{\text{cluster}} \),
where \( t_{\text{cluster}} \) is the time the request spends inside the cluster. In the remainder of
the paper, the term response time refers to \( t_{\text{cluster}} \).

Cluster Services and Utility Functions

The objective of the system is to maximize a cluster utility function that measures
how well the system meets the performance targets for the services it is offering.

In our design, we associate two performance targets with each service: the max-
imum response time (defined per individual request), and the maximum rejection
rate (defined over all the requests for a specific service).

For each service, a utility function specifies the rewards for meeting and the penalties for missing the performance targets. Let \( \rho \) denote the maximum allowed rejection rate and \( r \) represent the currently experienced rejection rate.

Meeting or exceeding the QoS objectives for a service yields a reward:

\[
U^+ = \alpha(\rho - r), \text{ if } r \leq \rho.
\]

Violating the QoS objectives for a service results in a penalty:

\[
U^- = -\alpha(r - \rho), \text{ if } \rho < r \leq \rho + \alpha^{1/(\beta-1)}.
\]

In order to avoid starvation of a service, the penalty increases exponentially when \( r > \rho + \alpha^{1/(\beta-1)} \);

\[
U^- = -(r - \rho)^\beta, \text{ if } r > \rho + \alpha^{1/(\beta-1)}.
\]

Fig. 7.3 shows the graph of such a service utility function. The control parameters \( \alpha \) and \( \beta \) are positive numbers that affect the shape of the service utility function and are used in our design to define the relative importance of a service.

We define the cluster utility function as the sum of the service utility functions of the services offered by the cluster. The goal of the system design is to maximize continuously this function. Consequently, the system design must attempt that, in case of load fluctuations, overload or server failures, cluster resources are allocated in such a way that the performance targets of services with higher values for \( \alpha \) and \( \beta \) are more likely to be met than those of services with lower values for those parameters.
CHAPTER 7. A MIDDLEWARE DESIGN FOR LARGE-SCALE CLUSTERS
OFFERING MULTIPLE SERVICES

System and Service Overlays

The servers of a cluster with $s$ service classes form $s + 1$ overlay networks: one system overlay and $s$ service overlays, one per service class. In our design, each server runs, at any time, one service $S$ out of these $s$ services. A server is a member of two overlay networks, the system overlay and the overlay for the service $S$.

A service overlay connects the servers of the same service class. Table 7.1 shows the structure of the service neighborhood table, which each server of this overlay locally maintains. An entry of this table corresponds to a neighboring server and has three fields: the server identifier, a timestamp with the local time of the latest update, and a data field containing state information: the utilization and the local rejection rate of that server.

The utilization is a binary value that takes the value 'light' if the utilization is below a threshold $max\_util$, and 'heavy' otherwise. We choose $max\_util$ in the linear region of the CPU utilization, increasing the probability that the system reacts in a linear fashion to the changes in the external load and that the servers do not become saturated.

The information in the service neighborhood table is used for request routing and for computing statistics that are advertised in the system overlay.

Apart from being a member of a specific service overlay, each node is also a member of the system overlay. The system overlay provides a common communication medium for all the servers of the cluster. Table 7.2 shows the structure of

![A service utility function.](image)

Figure 7.3: A service utility function.
Table 7.1: Structure of the service neighborhood table.

<table>
<thead>
<tr>
<th>Server ID</th>
<th>Timestamp</th>
<th>Utilization</th>
<th>Rejection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>id₁</td>
<td>ts₁</td>
<td>u₁</td>
<td>r₁</td>
</tr>
<tr>
<td>id₂</td>
<td>ts₂</td>
<td>u₂</td>
<td>r₂</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>idₙ</td>
<td>tsₙ</td>
<td>uₙ</td>
<td>rₙ</td>
</tr>
</tbody>
</table>

Table 7.2: Structure of the system neighborhood table.

<table>
<thead>
<tr>
<th>Server ID</th>
<th>Timestamp</th>
<th>Utilization</th>
<th>Rejection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>id₁</td>
<td>ts₁</td>
<td>service₁</td>
<td>r₁</td>
</tr>
<tr>
<td>id₂</td>
<td>ts₂</td>
<td>service₂</td>
<td>r₂</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>idₙ</td>
<td>tsₙ</td>
<td>serviceₙ</td>
<td>rₙ</td>
</tr>
</tbody>
</table>

Table 7.3: System management table.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Last Update</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity</td>
<td>ts_conn</td>
<td>conn</td>
</tr>
<tr>
<td>Cycle Length</td>
<td>ts_cycle</td>
<td>Δt</td>
</tr>
<tr>
<td>Service 1 Description</td>
<td>ts_service₁</td>
<td>sid₁, α₁, β₁, ρ₁, τ₁</td>
</tr>
<tr>
<td>Service 2 Description</td>
<td>ts_service₂</td>
<td>sid₂, α₂, β₂, ρ₂, τ₂</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Service n Description</td>
<td>ts_servicen</td>
<td>sidₙ, αₙ, βₙ, ρₙ, τₙ</td>
</tr>
</tbody>
</table>

the system neighborhood table. An entry in this table corresponds to a neighboring server and has four fields: the server identifier, a timestamp with the local time of the latest update, and a data field containing state information: the class of service that the neighbor currently provides and the measured rejection rate for that class of service. The service selection mechanism uses the information in this table.

Local Tables for Management

A server maintains locally three management tables: the system management table for the system overlay (Table 7.3), the service management table for the specific service it runs (Table 7.4), and the local state management table (Table 7.5). Each
entry in one of these tables contains information about a specific management parameter: its name, its value, and the timestamp of its last update. The connectivity of an overlay represents the number of neighbors of a node in that overlay. The cycle length indicates the frequency at which the mechanisms of overlay construction and service selection run. The description of each service contains the service id, the control parameters \( \alpha \) and \( \beta \), the maximum rejection rate \( \rho \) and the maximum response time \( \tau \).

We explain below how the service selection mechanism uses the information stored in the service descriptions fields, together with the information in the system neighborhood table, to estimate the current cluster utility and predict possible future utility values.

In \[98\], we have shown (a) how to monitor the average response time and the size of our system and (b) how to control the system behavior by changing the value of the connectivity parameter.

**Server Functionality**

Each server runs two local mechanisms - the application service that processes the requests and the local admission controller that observes the QoS objectives and decides, for each incoming request, whether the node should schedule and process it locally, or whether it should redirect it. Every server also runs instances of the three decentralized mechanisms that control the behavior of the system: topology construction, request routing and service selection.
7.2. SYSTEM DESIGN

Decentralized Control Mechanisms

The topology construction mechanism maintains the system and service overlays by integrating new nodes and by adjusting the topology when nodes depart or fail. It uses Newscast, an epidemic algorithm ([39]), to assign periodically a new set of logical neighbors to each server. At the same time, it updates the knowledge that a node has about the state of its neighbors through the neighborhood tables. The fact that servers exchange states only with their logical neighbors contributes to the scalability of the design.

Newscast produces a directed overlay with a configurable connectivity $c$ and with small-world properties (see discussion in the related work section).

The topology construction mechanism maintains the neighborhood tables as follows. Periodically, a server exchanges its neighborhood table with a randomly chosen neighbor. Together with the table, the server sends an additional entry with information about its current state. After the exchange, each server rebuilds its neighborhood table by selecting the $c$ neighbor entries with the most recent timestamps from the union of the two original neighborhood tables. This way, both nodes end up with identical tables. Note that the use of timestamps gradually erases old state information from the neighborhood tables.

After a node rebuilds its neighborhood table, it contacts each of its neighbors in order to update the state fields in the table. The management tables are updated in the same way.

Two control parameters from the management table can influence the behavior of the topology construction mechanism: the connectivity parameter $c$ and the duration $\Delta t$ between two successive runs of the mechanism on a node.

The topology construction mechanism runs not only on servers, but also on the entry points. An entry point runs this mechanism for each service overlay. This way, the entry point receives periodically the ids of a set of servers that provide each of the services the cluster offers. (Entry points do not send information about their current state to neighbors, to avoid requests being forwarded to them.)

The request routing mechanism directs service requests along the service overlays toward available resources, subject to QoS objectives. In this way, load balancing among servers of the same service is achieved. Service requests do not have a predetermined destination server; any server with sufficient resources can process a request. A server bases its routing decisions on its local state and the states of its neighbors in the service overlay.

The routing mechanism is invoked when the admission controller rejects a request due to insufficient local resources. The node checks its neighborhood table for light neighbors (with utilization below a threshold $\text{max\_util}$). If the node has light neighbors, it picks one of them at random and forwards the request to it. Otherwise, it picks at random any neighbor and sends the request to it. In order to avoid routing loops, each request keeps in its header the addresses of the servers it has visited.
Table 7.6: The execution cycle of the service selection mechanism.

1. Service id my_service, s, new_service;
2. Hashtable known_services;
3. Double max_util;
4. while (true) {
5.   for each s in known_services {
6.     estimateCurrentRejectionRate(s);
7.     predictRejectionRateAfterSwitch(s);
8.   }
9.   max_util = estimateNhoodUtility(my_service);
10.  new_service = my_service;
11.  for each s in known_services {
12.     U_s = predictNhoodUtilityAfterSwitchTo(s);
13.     if (U_s > max_util) {
14.        max_util = U_s;
15.        new_service = s;
16.     }
17.  }
18.  if (new_service != my_service) {
19.     if (local_utilization < random(0%, 100%)) {
20.        switchService(new_service);
21.        my_service = new_service;
22.     }
23.  }
24.  wait(delta_t);
25. }

The routing mechanism rejects a request from the cluster if (a) a node cannot process the request and the request has already visited all of its neighbors, or (b) the request has been redirected so many times that its maximum response time objective can no longer be met.

In the scenarios we have investigated for our design, we have found that typically a few percent of the requests are redirected when the system is lightly loaded.

The service selection mechanism partitions the system resources between services. Each server locally estimates the utility of its neighborhood and compares that to the utility the neighborhood would generate in the event the node would switch to a different service. The server then probabilistically switches to the service it predicts will yield the highest utility for the neighborhood.

This mechanism uses state information from the system neighborhood and the management tables, which are updated by the topology construction mechanism.

Each server starts the service selection mechanism at a random time and per-
forms the control loop in Table 7.6 with period $\Delta t$. The server estimates the current rejection rate for each service it has knowledge of (line 6). Then, for each service, the server predicts the rejection rate for the event it would switch to that service (line 7). After that, the server uses the estimated rejection rates to estimate the current neighborhood utility (line 9). The server then uses the above estimated and predicted rejection rates to predict the neighborhood utility for the event of switching to each service it knows about (line 12).

The server probabilistically switches to the service that maximizes the predicted neighborhood utility. The probability for switching increases with decreasing server utilization (line 19). We choose a probabilistic approach, because applying a greedy policy (deterministic switching) can lead to oscillations, where several servers periodically switch between two or more services.

7.3 System Evaluation Through Simulation

We implemented our design in Java and studied its behavior through simulation. We use JavaSimulation, a Java package that supports process-based discrete event simulation [17]. We implemented four types of simulation processes: cluster, request generator, entry point and server. The cluster starts the simulation and creates instances of the request generator, the entry point and the servers. The request generator simulates arrivals of service requests at the entry points, which forward them to servers. Each server process runs the local and the distributed mechanisms described in the previous section. An entry point process runs the topology construction and the request routing mechanisms.

We measure three metrics for each service: the rate of rejected requests, the experienced average response time, and the number of servers assigned to the service. Note that the rate of rejected requests and the average delay of the processed requests measure the experienced quality of service, while the maximum response time is guaranteed by design.

We measure the cluster performance through the cluster utility function, which expresses the capability of the system to conform to the performance targets specified in the Service Level Agreements (SLAs).

In order to assess the goodness of our solution with respect to maximizing the cluster utility, we compare the value of the cluster utility function against the maximum achievable utility of an ideal system. An ideal system has a processing capacity equal to the sum of the processing capacities of all the servers in the cluster. In addition, it has the following characteristics: (a) it is centralized and thus has no redirection delays and knows in real-time its state; (b) it has advance knowledge of the load pattern and allocates its resources optimally. Property (a) ensures that the ideal system does not lose any requests if the arrival rate is always smaller than the service rate. Property (b) ensures that the ideal system selectively drops requests to maximize the utility function whenever the arrival rate is higher than the service rate.
CHAPTER 7. A MIDDLEWARE DESIGN FOR LARGE-SCALE CLUSTERS
OFFERING MULTIPLE SERVICES

Simulation Scenarios

Four scenarios study the cluster behavior in case of (a) medium external load with minor fluctuations; (b) medium load followed by sudden overload, (c) increasing the relative importance of a service during overload, and (d) server failures leading to the overload of the remaining servers.

The simulated system has one entry point and 400 servers. In each overlay (system or service) a node has 20 logical neighbors. Each server has an execution cycle of 5 sec. It can process concurrently 10 requests, each with an execution time of 250 ms. Consequently, the service rate of a node is 40 req/sec and that of the cluster is 16000 req/sec.

The system provides three services \((S_1, S_2, S_3)\). Table 7.7 shows the QoS objectives and the control parameters for each service.

Table 7.7: Service QoS objectives and control parameters.

<table>
<thead>
<tr>
<th></th>
<th>QoS Objectives</th>
<th>Control Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max Resp Time (sec)</td>
<td>Max Rej Rate (%)</td>
</tr>
<tr>
<td>(S_1)</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>(S_2)</td>
<td>2.5</td>
<td>2</td>
</tr>
<tr>
<td>(S_3)</td>
<td>3.5</td>
<td>10</td>
</tr>
</tbody>
</table>

We model the request arrivals as a Poisson process. Each service has the same average arrival rate. We start measuring the output metrics after a warm-up phase of 100 sec. The graphs in Figs. 7.4-7.7 have been computed from averages of eight simulation runs.

Handling Minor Traffic Fluctuations

The *steady load* scenario evaluates the stability of the system under minor fluctuations in load and its conformance to the QoS objectives. The average external load is constant at 12000 req/sec or 75% of the total cluster capacity. Fig. 7.4 presents the results for this scenario.

Handling Overload

The *overload* scenario evaluates the capability of the system to dynamically reconfigure and continuously maximize the utility function in case where not all the QoS objectives can be met at the same time. The system starts with an average external load of 8800 req/sec or 55% of its total capacity. After 200 sec, the average external load rises to 17000 req/sec or 110% of the system capacity. Fig. 7.5 shows the results of this scenario.
7.3. SYSTEM EVALUATION THROUGH SIMULATION

![Graphs](image)

(a) rejection rate  
(b) average response time  
(c) server allocation  
(d) system utility

Figure 7.4: Results for the steady load scenario.

System Manageability

The manageability scenario evaluates the capability of the system to reconfigure in response to the system administrator changing a policy. The average external load is constant at 17600 req/sec, or 110% of the total cluster capacity. After 200 sec, the administrator increases the relative importance of $S_3$ by changing its control parameter $\alpha$ from 80 to 400. Fig. 7.6 shows the results of this scenario.
Handling Failures

The failure scenario evaluates the capability of the system to control the degradation of service quality in the case where a set of servers fails. The average external load is constant (10560 req/sec, or 66% of the total system capacity). After 200 sec, 40% of the servers fail at the same time. As a consequence, the external load on the functioning part of the system increases to 110%. Fig. 7.7 shows the results of this simulation scenario.
7.4 Discussion of the Results

The results of the simulation scenarios presented in Figs. 7.4-7.7 show that our design partitions the cluster resources according to the QoS objectives for each service, as long as the system has sufficient capacity to handle the external load. In case of overload, the resources are allocated according to the relative importance of each service.

The settling time of the system in response to external events varies. The system adapts fast to minor fluctuations in load, while it takes some 150 sec or 30 control cycles to fully stabilize after a sudden increase in load or a massive failure, leading
CHAPTER 7. A MIDDLEWARE DESIGN FOR LARGE-SCALE CLUSTERS
OFFERING MULTIPLE SERVICES

Figure 7.7: Results for the failure scenario.

to overload. We observe the same settling time of about 150 sec after changing a
management parameter in an overload setting.

Specifically, in the steady load scenario, Fig. 7.4(a) shows that minor fluctua-
tions in traffic lead to significant rejections only for the least important service $S_3$.
Fig. 7.4(b) shows that the average response time is about 1 sec for $S_1$, 2 sec for
$S_2$ and 3 sec for $S_3$, each of which is 0.5 sec better than the QoS objective. As
Fig. 7.4(c) shows, the system assigns resources to services according to their order
of importance. Fig. 7.4(d) shows that the system utility comes very close to the
utility of an ideal system.
7.4. DISCUSSION OF THE RESULTS

![Distribution of response times](image)

Figure 7.8: Distribution of the response times; $\tau_i$ is the maximum response time for service class $i$.

In the overload scenario, Fig. 7.5(a) shows that the rejection rates are virtually zero for low utilization, but increase considerably in overload, as expected. While the average rejection rate in the ideal system is 10%, the rejection rate in our system is about 17%. Specifically, the rejection rates are about 5% for service $S_1$, 20% for service $S_2$ and 26% for service $S_3$. (Note that the break points according to Fig. 7.3 occur for the following rejection rates: 10% for service $S_1$, 22% for $S_2$ and 90% for $S_3$). Fig. 7.5(b) shows that the average response time increases after the system becomes overloaded. An interesting observation in Fig. 7.5(c) is that, for a low load, the system allocates more servers to service $S_3$ than to service $S_2$. We explain this by the fact that, if the load is low and the rejection rates are close to zero, servers tend to stay with their current service and will not switch to a different service. However, the system assigns resources to services in their order of importance, as soon as the overload occurs. Finally, Fig. 7.5(d) shows that the utility is close to the utility of an ideal system during low load, but drops significantly when a large increase in the load occurs. It then converges towards a value closer to the ideal utility as the system reconfigures in response to the external event.

In the manageability scenario, Fig. 7.6 shows that after the control parameter of service $S_3$ has been increased, the rejection rate for $S_3$ decreases, as intended. This shows the capability of the system to enable an administrator to control, under conditions of overload, the relative importance and therefore the QoS degradation experienced by each service.

In the failure scenario, presented in Fig. 7.7, the metrics show a system behavior similar to that encountered in the overload scenario. Interestingly, the cluster utility
tends to be higher after the failure (failure scenario) than after the load increase (overload scenario).

Exponentially Distributed Execution Times

The simulation results presented in Section 7.3 are based on the assumption that each request has a deterministic execution time of 250ms. In today’s server clusters, however, the execution times of service requests vary. To address this fact, we modeled the execution times of the requests using an exponential distribution with the probability distribution function \( f(x) = \lambda e^{-\lambda x} \). For simulation experiments, we chose \( \lambda = 4 \), which means that the average execution time of a service request is \( 1/\lambda = 250\text{ms} \), as in the deterministic case. All service classes have the same \( \lambda \) parameter.

We support probabilistic execution times in our design as follows. As in the deterministic case, a scheduler allocates resources for 250ms to each request, following a FIFO discipline. A request scheduled for local execution is processed and never dropped, even if its execution time turns out to be longer than 250ms. A key difference between using deterministic and probabilistic execution times is that the maximum response time objective is always met, by design, in the deterministic case, but is violated for a subset of requests in the probabilistic case. From the assumed distribution of the execution times, we know that 37% of the requests will have an execution time larger than 250ms. (The probability of a request execution lasting less than the time \( \tau \) is: \( P\{t \leq \tau\} = 1 - e^{-\lambda \tau} \). The probability that a request executes for a time longer than \( 1/\lambda = 250\text{ms} \) is therefore \( 1/e \approx 0.37 \).

We have implemented the design with the above probabilistic execution times and ran the scenarios described in Section 7.3 with the same input parameters. In the following, we report results from the failure scenario and compare them with the simulation results from Section 7.3.

The new traces for the failure scenario are very similar to the ones in Fig. 7.7, and the measured metrics (rejection rates, average response times, server allocation and system utility) have values very close to those in Fig. 7.7.

Figs. 7.8(a) and 7.8(b) show the distribution of the response times for the deterministic and probabilistic execution times for the first 200 seconds of the failure scenario, before the failure occurs. The bin size for the measurements is 10ms. The vertical dotted lines mark the response time objectives for the service classes 1, 2 and 3.

Fig. 7.8(a) (deterministic case) shows two peaks in the distributions of the response times for each class: the first peak corresponds to requests processed on the first server immediately after being received. The second peak corresponds to requests processed in the cluster just before the deadline, i.e., the maximum response time objective. Fig. 7.8(b) shows the distribution of the response times for the case where the execution times are exponentially distributed.

We observe the following differences in the response time distributions: in the deterministic case, there is a peak in the distribution immediately before the re-
response time objective, for each service class. In the probabilistic case, the peak in each distribution occurs earlier than in the deterministic case, some 250ms before the response time objective, for each service class. In this case, each distribution stretches beyond the response time objective, and a fraction of the requests will thus violate this objective. Note that the average execution time is the same in both cases.

In conclusion, our evaluation suggests that the values of several output metrics of the system (rejection rates, average response times, server allocation and system utility) do not significantly change, when moving from the deterministic to the probabilistic case, as long as the average processing time stays the same. The maximum response time, however, is different. When using an exponential distribution, some requests will always violate the maximum response time objective. Note though that one can control the tradeoff between resource consumption and the fraction of requests that violates the response time objectives, e.g., by adjusting the time that the scheduler allocates to each request for execution.

7.5 Related Work

Various aspects of our design relate to platforms for web services with QoS objectives, peer-to-peer systems, applications of epidemic protocols, activities within the grid computing community, and systems controlled by utility functions.

Centralized Management of Web Services with QoS Guarantees

Today's server clusters follow a three-tiered architecture. A key element of such architecture is the Layer 4 switch [22], through which requests enter the cluster. While Layer 4 switches provide load balancing functionality, their design becomes increasingly complex for clusters with QoS objectives and a large number of servers or applications.

Several other studies address the problem of managing several web services with QoS objectives on a single cluster. In [27] and [28], the authors propose centralized resource management schemes for balancing service quality and resource usage. Both approaches attempt to maximize dynamically a utility function under fluctuating load. As in our design, these approaches map service requests into service classes, whereby all requests in a given service class have the same QoS objective. In [24], a scheduler with dynamic weights that controls overload in web servers is proposed. In [29], the authors present an architecture in which the central dispatcher of an overloaded data center redirects requests to geographically remote but less loaded data centers.

What fundamentally distinguishes our design from these approaches is that our design is decentralized, and all our cluster components are of the same type. We believe that our approach leads to a lower system complexity and thus the task of configuring the system becomes simpler. In addition, it has no single points of failure.
CHAPTER 7. A MIDDLEWARE DESIGN FOR LARGE-SCALE CLUSTERS
OFFERING MULTIPLE SERVICES

Structured Peer-to-Peer Systems

Our design shares several principles with peer-to-peer systems. After having studied the possibility of developing our architecture on top of a structured peer-to-peer system, we concluded that such an approach would likely lead to a system that is more complex and less efficient than our design, and we explain here briefly why. (To keep the term short, we use peer-to-peer system instead of structured peer-to-peer system.)

Peer-to-peer systems are application-layer overlays built on top of the Internet infrastructure. They generally use distributed hash tables (DHTs) to identify nodes and objects, which are assigned to nodes. The primary service of a peer-to-peer system is to route a request with an object identifier to a node that is responsible for that object. Routing is based on the object’s identifier and most systems perform routing within \( O(\log(n)) \) hops, where \( n \) denotes the system size. Routing information is maintained in form of a distributed indexing topology, such as a cube or a hypercube, which defines the topology of the overlay network.

Even though peer-to-peer networks efficiently run best-effort services ([48], [49], [50], [51]), no results are available to date on how to achieve service guarantees and service differentiation using peer-to-peer middleware. If one wanted to use a peer-to-peer layer as part of the design of a server cluster, one would assign an identifier to each incoming request and would then let the peer-to-peer system route the request to the node responsible for that identifier. The node would then process the request. In order for the server cluster to support efficiently QoS objectives, some form of resource control or load balancing mechanism would be needed in the peer-to-peer layer.

Introducing load-balancing capabilities in DHT-based systems is a topic of ongoing research ([52], [53], [54], [55]). An interesting result (as discussed in [52]) is that, in a network with \( n \) nodes, uniform hashing by itself results in an \( O(\log(n)) \) imbalance in the number of objects assigned to a node. Recently proposed solutions to the problem of load balancing in DHT systems include “the power of two choices” [52], or load stealing schemes and definition of virtual servers ([53], [54]).

In order to implement an efficient resource allocation policy that dynamically adapts to external load conditions, the identifier space in a peer-to-peer system needs to be re-partitioned and the partitions reallocated on a continuous basis. This means that the indexing topology, a global distributed state, needs to be updated continuously to enable the routing of requests to nodes.

When comparing the overhead associated with request routing based on DHTs with the routing mechanism in our design, we concluded that maintaining a global indexing topology is significantly more complex than maintaining the local neighborhood tables in our design.

In addition, peer-to-peer systems have properties that are not needed for our purposes. For instance, a peer-to-peer system routes a request for an object to a particular server - the one that is responsible for that object. (This property is useful to implement information systems on peer-to-peer middleware.) In our design, a
7.5. RELATED WORK

request can be directed to any server with available capacity, which simplifies the routing problem in the sense that there is no need to maintain a global distributed state.

Epidemic Protocols for Constructing Overlays

Epidemic protocols, also called gossip protocols, have been proposed to disseminate information in large-scale systems. In epidemic protocols, a node asynchronously exchanges local information with some neighbor on a periodic basis, which allows to build scalable and robust systems with low communication overhead [99].

The use of epidemic protocols for constructing overlays has been proposed in the context of tasks such as data aggregation, resource discovery and monitoring [13], database replication ([36], [57]), and handling web hotspots [58].

In our design, we apply an epidemic algorithm, Newscast [59], to locate available resources in a system that is subject to rapid changes. The topology construction mechanism uses Newscast to construct and maintain the overlays, through which requests are routed. We further use Newscast as a basis for disseminating control parameters.

Note that, while our design is independent of a specific epidemic protocol, the choice of the protocol will affect its performance. The properties of the overlay topology of the cluster derive directly from the properties of the underlying epidemic protocol. Newscast creates a directed overlay with uniform out-degree and small-world properties. An alternative to Newscast would be CYCLON [15], another epidemic algorithm that builds directed overlays with random-graph properties, i.e., low diameter and a uniform in-degree.

Resource Allocation in Grid and Utility Computing

Resource management and task scheduling in distributed environments is a key topic in grid computing and utility computing. The goal is often to maximize a given workload subject to a set of performance objectives. Many activities in grid computing focus on statistical prediction models for resource availability ([60], [61], [62]). These models are used as part of centralized scheduling architectures, in which a single scheduler dispatches tasks to a set of available machines. Other research creates peer-to-peer desktop grids that, however, do not work under QoS objectives [64].

Quartermaster [79] is an integrated set of modules that provides resource management for utility computing systems. It ensures a consistent system configuration, dynamically provisions resources for applications and optimizes the usage of these resources at runtime. Contrary to our approach, Quartermaster is based on a centralized architecture. Moreover, our design controls the system based on explicit QoS objectives, while Quartermaster is driven by the resource needs of specific applications.
CHAPTER 7. A MIDDLEWARE DESIGN FOR LARGE-SCALE CLUSTERS
OFFERING MULTIPLE SERVICES

Utility Functions to Control System Behavior

The idea of controlling the behavior of a computer system using utility functions has been explored in a variety of fields: scheduling in real-time operating systems, bandwidth provisioning and service deployment in service overlay networks, learning algorithms in artificial intelligence, and server clusters.

Time utility functions (TUF) have been defined in the context of task scheduling in real-time operating systems [66]. Given a set of tasks, each with its own TUF, the goal is to design scheduling algorithms that maximize the accrued utility from all the tasks. The authors discuss a wide range of possible time utility functions.

In [67], the authors apply time utility functions to design a scheduling framework for a real-time switched Ethernet.

The bandwidth-provisioning problem for a service overlay network (SON) has been modeled as an optimization based on utility functions, for example in [68]. The problem is to determine the link capacities that can support QoS sensitive traffic for any source-destination pair in the network, while minimizing the total provisioning costs.

In [76], the authors propose a decentralized scheme for replicating services along an already established service delivery tree. This algorithm estimates a utility function in a decentralized way; however, this process takes place along a fixed service delivery tree.

In the area of machine learning, reinforcement-learning algorithms use utility functions to model the impact of system components on the surrounding environment, as well as the feedback provided by the environment to the learning algorithm [70].

Apart from [76], we could not find any system design in the literature where a utility function is evaluated in a decentralized way, which is a key feature of our design.

7.6 Discussion and Future Work

This paper focuses on introducing self-management capabilities into the design of cluster-based services. It aims at making service platforms dynamically adapt to the needs of customers and to environment changes, while giving the service providers the capability to adjust operational policies at run-time.

Contribution

The key contributions of this paper are as follows. We have provided experimental proof that, with three decentralized mechanisms, one can design a middleware for web services with QoS objectives. A system following our design is self-organizing, scalable, robust and manageable. We demonstrate that epidemic protocols can be used to disseminate states and control parameters in a scalable and robust way. We show how to apply utility functions to express global objectives for service
7.6. DISCUSSION AND FUTURE WORK

differentiation and how to compute them in a decentralized fashion. Finally, we show, within our engineering parameters, that the design is efficient by comparing it to an ideal system.

Note that other types of utility functions could be used with our design. In this paper, we considered two objectives for each service, the maximum response time and the maximum rejection rate. We defined the utility for each service as a function of its rejection rate. Alternatively, a service utility could be defined as a function of the average response time, for example.

Our experience shows that one must carefully choose the parameters \( \alpha \) and \( \beta \) for the utility function. For instance, choosing a very large \( \alpha \) for one service and a very low \( \alpha \) for another service can result in virtually all the system resources being allocated to the service with the high \( \alpha \). Furthermore, the choice of \( \alpha \) and \( \beta \) for each service defines the order of the break points (see Fig. 7.3). This order will influence the way in which resources are partitioned between services when the system is in overload.

Current and Future Work

We have recently implemented the design presented in this paper in Java, as a filter in the Tomcat web server environment [19]. A key feature of this implementation is that it is non-intrusive, in the sense that it does not require any code modifications in Tomcat, nor of the underlying operating system (Linux or Windows).

We are currently studying the behavior of our middleware implementation on a cluster of web servers, using the TPC-W benchmark [20]. We use MySQL [100] as database software. The current testbed currently contains 15 PCs (two entry points, 10 application servers, a central database server and two generators of client sessions).

This work will allow us to validate further the design in a real environment. The testbed implementation will allow us to measure communication delays, execution times, server loads, etc., in well-defined TPC-W scenarios. Based on these measurements, we will be able to refine our simulation model to predict the behavior of the system for configurations much larger than the testbed. In addition, we plan to address additional issues, such as server affinity and requests that involve several system components.

The design we have presented in this paper can be extended in several ways, some of which we plan on pursuing. In [102], we proposed an alternative decentralized design that supports several applications on a single server, considers memory resources in addition to CPU resources and takes service migration costs into account. While compared to the design in this paper [102] scales better with the number of services, it does not allow managing the system through QoS objectives. We plan to engineer a system that combines the important features from both designs.
Chapter 8

Implementation and Evaluation of a Middleware for Self-Organizing Decentralized Web Services

Constantin Adam and Rolf Stadler
Laboratory for Communication Networks
KTH Royal Institute of Technology, Stockholm, Sweden
e-mail: ctin@kth.se, stadler@kth.se

Abstract

We present the implementation of Chameleon, a peer-to-peer middleware for self-organizing web services, and we provide evaluation results from a test bed. The novel aspect of Chameleon is that key functions, including resource allocation, are decentralized, which facilitates scalability and robustness of the overall system. Chameleon is implemented in Java on the Tomcat web server environment. The implementation is non-intrusive in the sense that it does not require code modifications in Tomcat or in the underlying operating system. We evaluate the system by running the TPC-W benchmark. We show that the middleware dynamically and effectively reconfigures in response to changes in load patterns and server failures, while enforcing operating policies, namely, QoS objectives and service differentiation under overload.

8.1 Introduction

Large-scale web services, such as on-line shopping, auctioning, and webcasting, rapidly expand in geographical coverage and number of users. Current systems
CHAPTER 8. IMPLEMENTATION AND EVALUATION OF A MIDDLEWARE FOR SELF-ORGANIZING DECENTRALIZED WEB SERVICES

that support such services, including commercial solutions (IBM WebSphere, BEA WebLogic) and research prototypes (Ninja [4], Neptune [5]), are based on centralized designs, which limit their scalability in terms of efficient operation, low configuration complexity and robustness.

To address these limitations, we have developed Chameleon, a decentralized middleware design that dynamically allocates resources to multiple services classes inside a global server cluster. Chameleon has three features characteristic of peer-to-peer systems. First, the server cluster consists of a set of functionally identical nodes, which simplifies the design and configuration. Second, the design is decentralized and the cluster dynamically re-organizes after changes or failures. Third, each node maintains only a partial view of the system, which facilitates scalability.

Chameleon supports QoS objectives for each service class. Its distinctive features are the use of an epidemic protocol [59] to disseminate state and control information, as well as the decentralized evaluation of utility functions to control resource partitioning among service classes. In [101], we have presented our design in detail and evaluated it through extensive simulations. This work complements simulation studies we have conducted earlier to validate the scalability of the Chameleon design. The implementation presented in this paper further validates the design and the system model we have used in our simulations.

The rest of this paper is structured as follows: Section 8.2 reviews the design of our peer-to-peer middleware. Section 8.3 describes the implementation of the design on an experimental test bed. Section 8.4 evaluates the implementation. Section 8.5 reviews related work. Finally, Section 8.6 contains additional observations and outlines future work.

8.2 Overview of the Chameleon Middleware Design

We have developed our design for large-scale systems. A typical deployment scenario that contains several data centers and many entry points is shown in Fig. 8.1. We assume that the data centers are connected through high-capacity links and the networking delays are much smaller than the processing delays. We refer to the collection of servers in these centers as the cluster.

Note that the design in this paper covers the tier of the application servers, but not the database tier. Specifically, we do not provide an approach to scale the database tier, which is an active area of current research. This is also reflected in the evaluation of the design, where we use database operations in browsing mode for read-only data, which do not have transaction requirements.

Service requests enter the cluster through the entry points, which function in our design as layer 7 switches. An entry point associates an incoming request with a service class and directs it to a server assigned to that class. An epidemic protocol runs in the cluster to provide the entry points with information about servers assigned to each service class.
8.2. OVERVIEW OF THE CHAMELEON MIDDLEWARE DESIGN

We have developed a design for large-scale, self-configuring server clusters. The design includes the servers in data centers, the entry points and a management station.

We associate two performance targets with each service: the maximum response time (defined per individual request), and the maximum drop rate (defined over all the requests for a specific service). In this paper, we use the terms service and service class interchangeably.

Upon receiving a request, a server schedules it for execution, using a FIFO policy. In case of overload, or high CPU utilization, the server redirects the request to one of its neighbors that runs the same service. A request that cannot be processed within the maximum response time objective is dropped by the system.

Cluster Services and Utility Functions

The objective of the system is to maximize a cluster utility function that measures how well it meets the performance targets for the offered services. We define the cluster utility function as the sum of the service utility functions. A service utility function specifies the rewards for meeting and the penalties for missing the performance targets for a given service.

Let $\rho$ denote the maximum allowed drop rate and $r$ represent the experienced drop rate. Meeting or exceeding the QoS objectives for a service yields a reward:

$$ U^+ = \alpha (\rho - r), \text{ if } r \leq \rho. $$

The violation of the QoS objectives for a service results in a penalty. In order to avoid starvation of a service, the penalty increases exponentially past the point $r^+ = \rho + \alpha^{1/(\beta-1)}$:
Figure 8.2: Three decentralized mechanisms control the system behavior: (a) topology construction, (b) request routing, and (c) service selection.

\[ U^- = \begin{cases} 
-\alpha(r - \rho), & \rho < r \leq r^+ \\
-(r - \rho)^2, & r > r^+ 
\end{cases} \]

The control parameters $\alpha$ and $\beta$ are positive numbers that define the shape of the graph and determine the relative importance of a service.

As the cluster utility is the sum of service utilities, the system will attempt to maximize the cluster utility by allocating cluster resources to services in such a way that the performance targets of services with higher values for $\alpha$ and $\beta$ are more likely to be met than those of services with lower values. This enables service differentiation in case of overload. Note that the behavior of a system depends on the specific choice of the cluster utility function. For example, a system for which the cluster utility function is defined as the minimum of the service utility functions will attempt to provide fair allocation to all cluster services.

**Decentralized Control Mechanisms**

Three distributed mechanisms, shown in Fig. 8.2, form the core of our design. Topology construction, based on Newcast, an epidemic protocol, organizes the cluster nodes into dynamic overlays, which are used to disseminate state and control information in a scalable and robust manner. Request routing directs service requests towards available resources, subject to response time constraints. Service selection dynamically partitions the cluster resources between services in response to external events, by periodically maximizing utility functions. These mechanisms run independently and asynchronously on each server.
8.2. OVERVIEW OF THE CHAMELEON MIDDLEWARE DESIGN

Table 8.1: Structure of the service neighborhood table.

<table>
<thead>
<tr>
<th>ID</th>
<th>Timestamp</th>
<th>Utilization</th>
<th>Processed Requests</th>
<th>Dropped Requests</th>
</tr>
</thead>
</table>

Table 8.2: Structure of the system neighborhood table.

<table>
<thead>
<tr>
<th>ID</th>
<th>Timestamp</th>
<th>Service</th>
<th>Processed Requests</th>
<th>Dropped Requests</th>
</tr>
</thead>
</table>

The topology construction mechanism organizes the servers of a cluster with \( s \) service classes into \( s + 1 \) logical networks: one system overlay and \( s \) service overlays, one per service class. Each server runs, at any time, a single service \( S \) and is a member of two logical networks, the system overlay and the overlay for the service \( S \). For each of these networks, a server has a neighbor table, with the structure shown in Tables 8.1 and 8.2. As we will explain below, the request routing mechanism uses the information in the service table, while the service selection mechanism uses the information in the system table.

The request routing mechanism directs incoming requests along the service overlays toward available resources, subject to the maximum response time objective. It is invoked when a server is overloaded and does not have sufficient local resources to process a request in time. If the server has light neighbors (i.e. with utilization below a threshold \( \text{cpu}_\text{max} \)) in its service neighborhood table, it forwards the request to a randomly selected light neighbor. Otherwise, it forwards the request to a random neighbor. In order to avoid routing loops, each request keeps in its header the addresses of the servers it has visited. If a node cannot process the request and the request has already visited all of its neighbors, the request is dropped.

The service selection mechanism partitions the system resources between services. Each server starts the service selection mechanism at a random time and performs an execution cycle with period \( \Delta t \). A server uses the information in its system table (Table 8.2) to estimate the utility currently produced by its neighborhood (line 5) and compares that to the utility the neighborhood would generate in the event the node would switch to a different service (lines 7-17). The server then probabilistically switches to the service it predicts will yield the highest utility for the neighborhood. The probability for switching increases with decreasing server utilization (line 19). We choose a probabilistic approach, because applying deterministic switching can lead to oscillations, whereby several servers periodically switch between two or more services.
CHAPTER 8. IMPLEMENTATION AND EVALUATION OF A MIDDLEWARE FOR SELF-ORGANIZING DECENTRALIZED WEB SERVICES

80 1. Service_id my_service, s, new_service;
2. Hashtable known_services;
3. Double max_util;
4. while (true) {
5.   max_util=estimateNhoodUtility(my_service);
6.   new_service=my_service;
7.   for each s in known_services {
8.     estimateCurrentDropRate(s);
9.     predictDropRateAfterSwitch(s);
10. }}
11. for each s in known_services {
12.   U_s = predictNhoodUtilityAfterSwitchTo(s);
13.   if(U_s > max_util) {
14.     max_util=U_s;
15.     new_service=s;
16.   }
17. }
18. if(new_service!=my_service) {
19.   if(local_utilization<random(0%.100%)) {
20.     switchService(new_service);
21.   my_service=new_service;
22. }
23. }
24. wait(delta_t);
25. }

Figure 8.3: The pseudo-code of the service selection mechanism.

8.3 Implementation: Integrating Chameleon into the Tomcat Framework

We have implemented our middleware in Java, within the Tomcat environment [19]. We have chosen Tomcat because it is simple and it is widely available as open source. We believe that Chameleon could be integrated as well with other web server environments that support application filters, such as IBM WebSphere.

To let the reader better understand the integration, we provide a short description of Tomcat; Fig. 8.4 shows its internal architecture. The components of the architecture are called containers. The service container has a set of connectors that handle the communication with clients on a specified port (80 or 8080 for HTTP, 443 for HTTPS, etc.). Upon receiving a request, a connector dispatches it to a thread from a pool of pre-allocated threads. The host container allows collocating several domain names on the same physical machine. A host container
8.3. IMPLEMENTATION: INTEGRATING CHAMELEON INTO THE TOMCAT FRAMEWORK

Figure 8.4: Tomcat conceptual objects.

includes several application containers. Finally, each application container holds the servlets needed to run it, as well as filters - objects that preprocess incoming requests and post-process outgoing responses for that application.

The Chameleon middleware is implemented as an application filter, as shown in Fig. 8.5. The filter instantiates when the Tomcat server starts the application(s) associated with the filter. The instantiation starts several threads that run the three decentralized control mechanisms (topology construction, request routing and

Figure 8.5: Integrating Chameleon with Tomcat. Left: request flow for a locally processed request. Right: request flow for a routed request.
CHAPTER 8. IMPLEMENTATION AND EVALUATION OF A MIDDLEWARE FOR SELF-ORGANIZING DECENTRALIZED WEB

82

service selection), as well as a load calculator that measures the CPU utilization on the local machine.

As shown in Fig. 8.5, the connector forwards incoming application requests to the routing mechanism that is running inside the filter. Request routing decides whether the server should process the request locally, forward it to another server, or drop it. This decision process works as follows. The routing mechanism reads the time when the request entered the cluster from the 'pragma' field of the HTTP header of the request. (All the servers inside the cluster synchronize their clocks using the Network Time Protocol (NTP) [103].) If the difference between the local time and the time when the request entered the cluster is larger than the maximum response time objective, the server drops the request and sends the client an error page. Otherwise, the routing mechanism queries the load estimator about the local CPU utilization. If the utilization is below a threshold $cpu_{max}$, routing sends the request to the local application servlets for processing, as shown in Fig. 8.5(a). If the utilization is above $cpu_{max}$, then the routing mechanism forwards the request to another server in the service neighborhood table, as shown in Fig. 8.5(b).

We choose the threshold $cpu_{max}$ in the linear region of the CPU utilization, where a linear increase in load results in a linear increase in the CPU utilization. This choice decreases the probability that the servers become overloaded. We measure the CPU utilization on an application server every 2 sec. To smooth out spikes in the CPU utilization, we compute the CPU utilization at a given time by averaging the last 5 utilization measurements (taken over the last 10 seconds).

We use a two-way architecture to forward requests, in which the entry points and the application servers act as TCP gateways ([104], [26]). This technique, also known as ‘request chaining’, works as follows. The client establishes a TCP connection with an entry point (EP). When the EP or a server forwards a request to another server, it opens a TCP connection and sends the request over this connection. The reply travels back from the server that handles the request to the client following the TCP connections opened during the request forwarding phase. Other request forwarding schemes include TCP splicing [26], where the entry points function as Network Address Translators and TCP handoff ([104], [26], [23]), where the entry points hand off to the servers the TCP connections they establish with the clients. We did not use these techniques, because they both require changes to the operating system on the entry points. In addition, TCP handoff requires changes to the operating system on the servers, as well.

8.4 System Evaluation

Testbed and Setup

The test bed that we use for the evaluation includes twelve application servers (with Celeron 1.7 GHz processors and 128 MB RAM), four entry points and a database server (each with a Pentium IV 2 GHz processor and 256 MB RAM), connected via
100Mbps Ethernet Switches, as shown in Fig. 8.6. We use four additional machines as client emulators for running the TPC-W benchmark, as described below.

We study the behavior of the system using TPC-W [20], a benchmark that emulates an electronic bookstore. We use the TPC-W implementation from the University of Wisconsin-Madison [165]. This implementation contains a set of Java servlets that run on the application servers and a request generator called Remote Browser Emulator (RBE). An important parameter of the RBE is the number of emulated browsers, which controls the amount of the generated load.

We generate different workloads with TPC-W and study the ability of the system to adapt to changes in external load, to failures, or the ability to provide service differentiation under overload conditions. Note that the usual TPC-W metric of Web Interactions per Second (WIPS) is not of primary interest in our evaluation.

An important property of the load pattern experienced by web services and captured by TPC-W is that the load depends on the state of the servers. If the servers are highly loaded, the request rate decreases. For instance, a response to a web request for a page with many images will generate additional requests, one request per image. In case the original request for the page is dropped or delayed, then the subsequent requests for the images on the page will be dropped or delayed as well.

The TPC-W servlets perform various operations, such as generating dynamic HTML pages, fetching images and reading from or writing to a database populated with 10000 items and 288000 customer entries. We use MySQL database software,
CHAPTER 8. IMPLEMENTATION AND EVALUATION OF A MIDDLEWARE FOR SELF-ORGANIZING DECENTRALIZED WEB SERVICES

Table 8.3: Service QoS objectives and control parameters.

<table>
<thead>
<tr>
<th>QoS Objectives</th>
<th>Control Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Resp. Time</td>
<td>Max Drop Rate</td>
</tr>
<tr>
<td>S_1</td>
<td>1 sec</td>
</tr>
<tr>
<td>S_2</td>
<td>3 sec</td>
</tr>
</tbody>
</table>

version 4.1.12.

Because our evaluation focuses on the application servers, we configure the TPC-W RBEs to generate traffic following the browsing mix that is specifically designed to load the application servers. This mix contains 95% browse activity and only 5% order activity, which has consistency requirements. The application servers run local databases that handle browsing database queries. Order requests are sent to a dedicated database server, which guarantees consistency.

On the test bed, we have configured the middleware to support two TPC-W service classes with different QoS objectives and control parameters, as shown in Table 8.3. Service 1 represents a premium service, while service 2 is the basic service, and the importance of service 1, as defined by the control parameters α and β is higher than that of service 2. Note that the TPC-W application contains fifteen different operations. In our setup, we assign the same QoS objectives and control parameters to each of these operations inside a single TPC-W service class.

The system is configured in such a way that each server has three neighbors in both the system and the service overlays. The upper threshold for the CPU utilization $cpu_{max}$ is set to 70%. Each server runs the service selection mechanism every 30 seconds.

During the measurements, we retrieve every 30 seconds four metrics for both service classes: the request arrival rate, the request drop rate, the average response time, and the number of servers assigned to each service. The request arrival rates and the average response times are measured on the entry points. The request drop rates and the number of servers assigned to each service are measured on the application servers.

Scenarios and Measurements

System Adaptability to Changes in the External Load: The first scenario tests the stability of the system under normal operating conditions and its capability to adapt to changes in the external load. The load pattern of this scenario develops as follows. The RBE launches gradually, over the first six minutes of the run, 1200 emulated browsers for service 1 and 800 emulated browsers for service 2. After 18 minutes, the RBE stops 200 emulated browsers for service 1 and launches an additional 200 emulated browsers for service 2. For the remainder of the run, 1000 emulated browsers are active for both service classes. Under this load, the system is
8.4. SYSTEM EVALUATION

Figure 8.7: System adaptability to changes in the external load.

not able anymore to meet the QoS objectives for the services and is thus overloaded
(see also Figs. 8.9(a), 8.9(b) and 8.9(c)).

Fig 8.7 shows the output metrics measured during a single run of this scenario.
During the first six minutes, the system operates under normal load, i.e., the system
could process more requests than the request arrival rate. As a result, the system
is stable in the sense that the servers do not switch service and the QoS objectives
are met for both service classes.

After the first six minutes, the arrival rate for service 1 increases and surpasses
the service capacity of the six servers that were initially assigned to service 1.
When this happens, the system starts reconfiguring and assigns more resources to
service 1 after some 2-3 execution cycles i.e., 1-1.5 minutes later. We explain this
delay first by the fact that the service utility function contains the drop rate as the
Chapter 8. Implementation and Evaluation of a Middleware for Self-Organizing Decentralized Web

only state variable. Therefore, the system must drop requests before its adaptation mechanisms become active. Second, once a server begins dropping requests, it takes some time before it advertises the drop rate to its neighbors. Finally, when a server learns that a neighbor is dropping requests, the server will only switch service probabilistically, based on its utilization. An overloaded server will switch services with very low probability.

After another few minutes, the system re-organizes in order to support the QoS objectives and provide preferential treatment to the premium service class during overload. We observe that the average response time (Fig. 8.7(a)) and the drop rate (Fig. 8.7(b)) are significantly lower for the premium service (service 1). At the same time, the number of servers assigned to service 1 is larger than the number of servers assigned to service 2 (Fig. 8.7(d)).

We see in Fig 8.7(c) that the arrival rate for service class 2 significantly decreases while the system is under overload. We explain this behavior by the dependence of the load pattern on the state of the servers, as described above.

System Reconfiguration after Failures: This scenario tests the capability of the system to reconfigure and continue providing service after server failures. The load pattern of this scenario develops as follows. The RBE starts a total of 800 emulated browsers for each service class. Each emulated browser runs for 30 minutes. The launches occur gradually over the first 15 minutes of the run. After 25 minutes, two failures occur, simulated by stopping Tomcat on two servers running service 1. Five minutes later, the traffic load starts gradually decreasing, as the emulated browsers complete their execution.

Fig. 8.8 presents the output metrics measured during a single run of this scenario. Fig. 8.8(d) shows that the system reconfigures by assigning more servers to service 1, which has a higher priority than service 2. The system starts reconfiguring after 2-3 execution cycles, i.e., 1-1.5 minutes. The reason for this delay is explained in the previous scenario.

System Performance as a Function of the External Load: In this series of experiments, we measure the capacity of the system, i.e. the maximum number of requests that the system can handle, for the case where both service classes share the same load. We also test the system capability to provide service differentiation under overload.

For this scenario, we use ten application servers. At the start of a run, five servers are assigned to each service class.

Each point in the graphs shown in Figs. 8.9(a), 8.9(b) and 8.9(c) contains the outcome of an entire run. In each run, we launch an equal number of emulated browsers for both classes and the load remains constant.

As expected, the service rate (in req/sec) increases linearly with the number of emulated browsers, up to a point, around 1000 emulated browsers, where the system starts dropping requests. Above 1400 emulated browsers, the system cannot meet the QoS objectives anymore, but still provides service differentiation in the sense that service 1 receives preferential treatment, which is reflected by lower drop rates and lower average response times. Around 1600 emulated browsers,
Figure 8.8: System reconfiguration after failures. The dashed line indicates the time when two servers fail.

the capacity limit is reached, above which the service rate of the system does not increase anymore. The total service rate remains flat, the service rate for service 1 slightly increases, while the service rate for service 2 slightly decreases.

Our measurement logs show that, up to a load of 1000 emulated browsers, a server does not change its service. Beyond that load, servers drop requests, which triggers service switches.
Figure 8.9: System Performance as a Function of the External Load.

8.5 Related Work

Today's server clusters follow a three-tiered architecture. A key element of such architecture is the Layer 4-7 switch [22], through which requests enter the cluster. While Layer 4-7 switches provide load balancing functionality, their design becomes increasingly complex for clusters with QoS objectives and a large number of servers or applications. Many works focus on the problem of efficiently dispatching requests into the server cluster. In [24], a scheduler with dynamic weights that controls overload in web servers is proposed. The weighted fair queue scheduler proposed in [25] uses an observation based adaptive scheme that increases the weight of a service class experiencing poor performance at the expense of another class that has more resource share and less demand. In [26], the authors compare various locally
8.6. DISCUSSION AND FUTURE WORK

distributed web system architectures. They provide a classification and analysis of various dispatching algorithms.

Several other studies address the problem of managing multiple web services with QoS objectives on a single cluster. In [27] and [28], the authors propose centralized resource management schemes for balancing service quality and resource usage. Both approaches attempt to maximize dynamically a utility function under fluctuating load. As in our design, these approaches map service requests into service classes, whereby all requests in a given service class have the same QoS objective. In [29], the authors present an architecture in which the central dispatcher of an overloaded data center redirects requests to geographically remote but less loaded data centers.

Addressing the interactions between the components of multi-tier systems is another research topic in the area of QoS for Web services. In [42], the authors develop an analytical model for multi-tier internet services. They propose using this model for tasks such as capacity provisioning, performance prediction, application configuration or request policing. In another work [43], the authors propose a mechanism that prevents overload and saturation of the database servers in small sized clusters, controlling the interaction between servlets and the database server.

8.6 Discussion and Future Work

In this paper, we present the implementation of Chameleon, a peer-to-peer middleware for self-organizing web services. We evaluate the system by running the TP C-W benchmark. We show that the middleware dynamically and effectively reconfigures in response to changes in load patterns and server failures, while enforcing operating policies, namely, QoS objectives and service differentiation under overload. This work complements simulation studies we have conducted earlier in [101] to validate the scalability of the Chameleon design.

Specifically, in the last evaluation scenario, the service rate of the system increases linearly with the offered load, up to the limit of its service capacity. While this can be expected in the operating region where the load is low and service switches do not occur, the linear dependency still holds, although with a smaller slope, while the system reconfigures and drops significant numbers of requests. Finally, in overload, the service rate remains constant, while the system still provides service differentiation.

A number of open issues require further consideration, as well. First, we do not address the so-called server affinity problem: our current design handles all service requests independently of one another and thus has no concept of a session. Second, we did not investigate modalities to decentralize the database tier. This is a very active research area, a number of approaches have been proposed (see for example [84]), and we will investigate the possibility of integrating our middleware with a distributed database architecture. Third, we do not perform request scheduling in the test bed implementation. Adding this capability to our implementation
CHAPTER 8. IMPLEMENTATION AND EVALUATION OF A MIDDLEWARE FOR SELF-ORGANIZING DECENTRALIZED WEB SERVICES may lead to a better control of the performance objectives. Request scheduling would require capabilities to do application profiling, as described in ([45], [46], [47]). Fourth, there exist several options to improve the performance of the request forwarding procedure. Finally, we plan to study the system behavior for several utility functions.

Acknowledgment

The authors would like to thank Prashant Varghese for his Master Thesis which studied the implementation design of Chameleon on a test bed. Readers interested in a deeper coverage of the implementation should also consult Prashant’s Master Thesis [106].
Chapter 9

A Decentralized Application Placement Controller for Web Applications

Constantin Adam\textsuperscript{1}, Giovanni Paci\textsuperscript{ci}\textsubscript{2}, Michael Spreitzer\textsuperscript{2}, Rolf Stadler\textsuperscript{1}, Malgorzata Steindler\textsuperscript{2}, Chunqiang Tang\textsuperscript{2}

\textsuperscript{1} School of Electrical Engineering, Royal Institute of Technology, Stockholm, Sweden
\textsuperscript{2} IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

Abstract

This paper addresses the problem of dynamic system reconfiguration and resource sharing for a set of applications in large-scale services environments. It presents a decentralized application placement scheme that dynamically provisions enterprise applications with heterogeneous resource requirements. Potential benefits, including improved scalability, resilience, and continuous adaptation to external events, motivate a decentralized approach. In our design, all nodes run a placement controller independently and asynchronously, which periodically reallocates a node’s local resources to applications based on state information from a fixed number of neighbors. Compared with a centralized solution, our placement scheme incurs no additional synchronization costs. We show through simulations that decentralized placement can achieve accuracy close to that of state-of-the-art centralized placement schemes (within 4\% in a specific scenario). In addition, we report results on scalability and transient behavior of the system.

9.1 Introduction

The service sector has undergone a rapid expansion in the past several decades. In the United States, services account for approximately three quarters of GDP
and eight out of ten jobs. This trend has driven the IT industry to shift its focus from the sales of computer hardware and software toward providing value-added IT services. Another trend in the industry is that many organizations increasingly rely on web applications to deliver critical services to their customers and partners. These organizations typically want to focus on their core competency and avoid unnecessary risks caused by sophisticated IT technologies. IT service providers are therefore encouraged to host the web applications in their data centers at reduced cost and with improved service quality.

An IT service provider that offers comprehensive enterprise computing services may run hundreds of data centers, and a single data center may host hundreds of applications running on thousands of machines. The sheer scale and heterogeneity of hardware and software pose grand challenges on how to manage these environments. Our vision is to engineer a middleware system that dynamically reconfigures in reaction to events, such as changes in the external demand, or node failures. In addition, the system should provide self-management functionality, by adapting to addition and removal of applications or resources, and changes in management policies. Fig. 9.1(a) is an example of such a system.

The idea of providing middleware support for large-scale web services is not new. Successful research projects in this area include Ninja [4] and Neptune [5], and leading commercial products include IBM WebSphere and BEA WebLogic. Despite the great success of these systems, the rapidly increasing demand for web services calls for next-generation middleware systems that address existing systems' limitations in resilience, scalability, and manageability.

Inspired by the recent success of peer-to-peer (P2P) systems with millions of users (e.g., KaZaA [107] and Skype [108]), we argue that next-generation middleware systems should leverage P2P technologies that have been proven to be scalable, fault-tolerant, and self-organizing. In P2P architectures, all active computing entities (or nodes) are treated equal. Nodes organize themselves into an application-level overlay network and collectively route traffic and process requests. Neither there is a single point of failure, nor the bottlenecks associated with a centralized system.

This paper focuses on a single component of our peer-to-peer middleware system: the application placement controller. Other components such as request routing and load balancing mechanisms will be presented in detail elsewhere. The placement controller supports several applications on a single node and therefore enables the deployment of a potentially large number of different applications in the system. Each node runs a placement controller that decides the set of applications that the node is offering. The decision aims at maximizing a global objective function.

The unique feature of our approach for application placement is the decentralized design of the placement controller. Each node independently and asynchronously executes the placement algorithm that manages its local resources. Although each node makes its own decisions independently, collectively all nodes work together to meet a global objective.

The problem of dynamic application placement has been studied before, and
centralized solutions have been proposed in [72, 79, 45]. We argue that our decentralized approach offers advantages over a centralized design, such as increased scalability, absence of single points of failure, zero configuration complexity, and the capability to adapt continuously to changes in demand and to failures. Compared to a centralized solution, the additional overhead incurred by our approach is small in terms of computational costs and less efficient use of resources.

The rest of the paper is organized as follows. Section 9.2 gives an overview of our P2P middleware system. Section 9.3 presents the details of our decentralized application placement algorithm. Section 9.4 evaluates the algorithm and compares it with state-of-the-art centralized algorithms. Related work is discussed in Section 9.5. Finally, Section 9.6 concludes the paper.

9.2 Overview of Our P2P Middleware System

Fig. 9.1(a) shows a possible deployment scenario for our system. We consider a large-scale system with several entry points, each of which is logically connected to a large number of nodes, potentially all the nodes in the system. An entry point directs incoming requests to the nodes using a forwarding policy, such as weighted round robin.

The nodes self-organize into a logical overlay network through application-level (TCP) connections. They use the overlay for exchanging state and routing information. Many overlay construction protocols [16, 7, 10, 8] have been proposed, and most of them could be used in our system. A comparative assessment of overlay construction mechanisms for our system is beyond the scope of this paper.

Upon receiving a request, a node either processes it locally or forwards it to a peer that offers that application.
CHAPTER 9. A DECENTRALIZED APPLICATION PLACEMENT CONTROLLER FOR WEB APPLICATIONS

Node Model

As shown in Fig. 9.1(b), a node runs a single application process for each application \((a_1, a_2, \ldots, a_m)\) it offers. The placement mechanism, which manages a set of applications, has three components: the profiler, the placement controller and the placement executor. The profiler gathers local application statistics, such as the request arrival rate, the memory consumed by an application process, and the average number of CPU cycles used by a request. The placement controller runs periodically on each node. It gathers application statistics from the local profiler, as well as from the profilers of the overlay neighbors. Then, it computes a list of application processes to run on the local node during the next placement cycle and sends this list to the executor, which stops and starts the required applications. The complete node model also includes components, such as request router, load balancer, and queue controller, which are not discussed here.

The Global Application Placement Matrix

If a node cannot process a request locally, it routes the request to another node using a local copy of the placement matrix \(P\), which lists for each application the nodes that run it.

In addition to supporting routing, \(P\) provides information, such as the number of instances of a given application that are running in the entire system. This data can be used by the application placement controller to implement policies, such as the minimum or maximum number of application instances in the system.

The local copy of the matrix \(P\) on each node is maintained by GoCast, a gossip-enhanced multicast protocol [16]. Whenever a node changes the set of applications running locally, it uses GoCast to disseminate the changes to all other nodes.

Note that maintaining a copy of the global placement matrix at each node potentially limits the scalability of the system, as the number of broadcast operations increases linearly with the system size. However, GoCast enforces a maximum message rate on each overlay link, as it buffers all the updates received by a node during a certain time interval and aggregates them into a single message. The limitation in scalability therefore stems from the fact that the size of an aggregated message and the processing load on a node increases with the system size.

9.3 Decentralized Application Placement Controller

The Application Placement Problem

We consider a set \(\mathcal{N}\) of nodes and a set \(\mathcal{A}\) of applications. Let \(n\) be a node in \(\mathcal{N}\) and \(a\) an application in \(\mathcal{A}\). An application’s demands for resources can be characterized as either load-dependent or load-independent. A running application instance’s consumption of load-dependent resources depends on the offered load. Examples of such resources include CPU cycles and disk bandwidth. A running
application instance also consumes some load-independent resources regardless of the offered load, even if the program processes no requests. An example of such a resource is the storage space for the program’s executable.

Due to practical reasons, we treat memory as a load-independent resource and conservatively estimate the memory usage to ensure that every running application has sufficient memory. Our system includes a component that dynamically estimates the upper limit of an application’s near-term memory usage based on a time series of its past memory usage. Because the memory usage estimation is dynamically updated, our placement controller indirectly considers some load-dependent aspects of memory.

We treat memory as a load-independent resource for several reasons. First, a significant amount of memory is consumed by an application instance, even if it receives no requests. Second, memory consumption is often related to prior application usage rather than to its current load. For example, even under low load, memory usage may still be high because of data caching. Third, because an accurate projection of future memory usage is difficult and many applications cannot run when the system is out of memory, it is reasonable to use a conservative estimation of memory usage, i.e., taking the upper limit instead of the average.

Among the many load-dependent and load-independent resources, we choose CPU and memory as the representative ones to be considered by our placement controller, because we observe that they are the most common bottleneck resources. For example, our experience shows that many business J2EE applications require on average 1-2GB real memory. While the description of our algorithm only considers CPU and memory, it can be applied to other types of resources as well. For example, if the system is disk-bound, we can use disk bandwidth instead of CPU cycles as the load-dependent resource. Regarding measuring the demand and resource consumption of an application, we refer readers to prior work [45, 46, 47] on application profiling.

For an application \( a \in \mathcal{A} \), let \( \gamma_a \) denote the memory demand of an instance of \( a \), and \( \omega_a^{req} \) denote the total CPU demand for \( a \) throughout the entire system. For a node \( n \in \mathcal{N} \), let \( \Gamma_n \) and \( \Omega_n \) denote its memory and CPU capacities, respectively. The CPU demand of an application is measured in CPU cycles/second (on a standard reference machine). We assume that, because of the memory constraint, a node \( n \) can run only a subset \( \mathcal{R}_n \) of all applications offered by the system. Let \( \omega^{req}_{n,a} \) be the CPU cycles/second needed on node \( n \) in order to process the request rate for application \( a \). Let \( \omega^{real}_{n,a} \) be the CPU cycles/second that node \( n \) allocates to application \( a \). Let \( \omega^{real}_a \) denote the total number of CPU cycles/second that the entire system allocates to application \( a \), i.e., \( \omega^{real}_a = \sum_{n \in \mathcal{N}} \omega^{real}_{n,a} \).

The goal of application placement is to maximize the sum of CPU cycles deliv-
CHAPTER 9. A DECENTRALIZED APPLICATION PLACEMENT CONTROLLER FOR WEB APPLICATIONS

We state the problem as follows:

\[
\max \sum_{n \in \mathcal{N}} \sum_{a \in \mathcal{A}} \omega_{n,a}^{real} \tag{9.1}
\]

such that

\[
\forall n \in \mathcal{N} \quad \Gamma_n \geq \sum_{a \in \mathcal{R}_n} \gamma_a \tag{9.2}
\]

\[
\forall n \in \mathcal{N} \quad \Omega_n \geq \sum_{a \in \mathcal{R}_n} \omega_{n,a}^{real} \tag{9.3}
\]

Formulas 9.2 and 9.3 stipulate that the allocated CPU and memory resources on each node cannot exceed the node’s CPU and memory capacities, respectively.

The Application Placement Algorithm

Our placement algorithm executes periodically on each node. The time between two executions of the algorithm is called the placement cycle.

The placement algorithm has three consecutive phases. In the first phase, a node gathers placement and load information from its neighbors. In the second phase, the node determines a set of applications to run locally during the next placement cycle. In the last phase, the node carries out the placement changes (i.e., start or stop of applications) and advertises its new placement configuration to other nodes using GoCast. We give the pseudo-code of the algorithm and describe each of its phases below.

1. class AppInfo {
   2.     string app_id;
   3.     double cpu_demand, cpu_supply;
   4. }
5. List<AppInfo> active_apps, standby_apps, new_active_apps;
6. List<AppInfo> neighbor_active_apps;
7. double max_cpu_supply, cpu_supply;

8. while(true) {
9.     active_apps=getLocalActiveApps();
10.    neighbors=overlay.getNeighbors();
11.    neighbor_active_apps=getAppStats(neighbors);
12.    standby_apps=getIdleAppStats(local_node, neighbors);
13.    for each app in neighbor_active_apps
14.        if(active_apps.contains(app)==false)

\textsuperscript{1}Our system can also be configured to optimize a certain utility function, but a detailed discussion is beyond the scope of this paper.
9.3. DECENTRALIZED APPLICATION PLACEMENT CONTROLLER

15. if(app.cpu_demand>app.cpu_supply)
16. standby_apps.add(app);
17. active_apps=sortIncreasingDensity(active_apps);
18. standby_apps=sortDecreasingUnmetDensity(standby_apps);
19. new_active_apps=active_apps;
20. max_cpu_supply=currentCpuSupply();
21. num_active_apps=active_apps.size();
22. for(i=0;i<num_active_apps;i++) {
23. remove top i apps from active_apps;
24. cpu_supply=allocateIdleResources(standby_apps);
25. if((cpu_supply-change_cost)>max_cpu_supply) {
26. max_cpu_supply=cpu_supply-change_cost;
27. new_active_apps=active_apps-top_i_active_apps+sel_standby_apps;
28. }
29. }
30. if(new_active_apps!="active_apps") {
31. advertise(new_active_apps, state=STARTING);
32. stopAndStartApps(active_apps, new_active_apps);
33. advertise(new_active_apps, state=ACTIVE);
34. }
35. wait until the end of the placement cycle;
36. }

Phase 1: Gathering State Information. A node retrieves from each neighbor \(x\) the list of applications \((a_1 \cdots a_m)\) running on \(x\), the memory requirements \((\gamma_1 \cdots \gamma_m)\) of those applications, the CPU cycles/second \((\omega_{x,a_1}^{\text{real}}, \cdots, \omega_{x,a_m}^{\text{real}})\) delivered to those applications, and the CPU demands of those applications \((\omega_{x,a_1}^{\text{req}}, \cdots, \omega_{x,a_m}^{\text{req}})\) (lines 10-11). In addition, neighbor \(x\) also reports the locally measured demands for applications it could not route, since they are not offered in the system (line 12). A high demand for these inactive applications might trigger their activation during the next placement cycle.

Phase 2: Computing a New Set of Active Applications. Using the information gathered in the previous phase, a node builds a set of active applications \(\mathcal{R} = \{r_1, \cdots, r_l\}\) (line 9) and a set of standby applications \(\mathcal{S} = \{s_1, \cdots, s_j\}\) (lines 12-16). \(\mathcal{R}\) contains the applications that are currently active on the local node. \(\mathcal{S}\) contains two types of applications: those that run in the neighborhood of the node but not on the node itself, and applications are currently not offered in the system.

The placement algorithm attempts to replace a subset of applications in \(\mathcal{R}\) with a subset of applications in \(\mathcal{S}\), so that the local CPU utilization is maximized. Since the candidate space for the optimal configuration grows exponentially with \(|\mathcal{R} \cup \mathcal{S}|\), we apply a heuristic that reduces the complexity of the problem to \(O(|\mathcal{R}| \times |\mathcal{S}|)\) (lines 17-29).

On a node \(n\), the active applications in \(\mathcal{R}\) are sorted in increasing order of their density \(d_n\), which we define as the load delivered by \(n\) to application \(a\), divided
by the memory consumption of \( a \), i.e., \( d_a = \omega_{n,a}^{\text{real}}/\gamma_a \). A high-density application consumes system resources efficiently, in the sense that it causes a relatively high CPU utilization while consuming a relatively little memory. The applications in the standby set \( S \) are sorted in decreasing order of their residual density \( d_a \), which we define as the unmet demand of \( a \) divided by its memory demand, i.e.,

\[
d_a^* = \sum_{n \in \text{neighbors}} (\omega_{n,a}^{\text{real}} - \omega_{n,a}^{\text{req}})/\gamma_a.
\]

The standby applications that have no unmet demands are removed from \( S \), because there is no need to start additional instances for them. Intuitively, the placement controller tries to replace low-density applications in the active set \( R \) with high-density applications in the standby set \( S \), so that the CPU utilization is maximized.

The placement algorithm has \(|R| + 1 \) iterations \((k = 0 \cdots |R|)\). During the first iteration \((k = 0)\), it does not remove any application from \( R \). If the local node \( n \) has available memory and CPU cycles (i.e., \( \Gamma_n^{\text{free}} > 0 \) and \( \Omega_n^{\text{free}} > 0 \)), then the algorithm attempts to add one or more applications from \( S \) to \( R \). This is done by selecting applications from the top of \( S \), subtracting the cost for starting the applications, and evaluating the resulting gain in CPU utilization.

During iteration \( k > 0 \), the algorithm removes the top \( k \) applications from \( R \). It then computes the available memory and CPU resources and attempts to assign these resources to applications in \( S \) in the following way. The algorithm attempts to fit the first application \( s_1 \in S \) into the available memory. If this operation succeeds, then the algorithm attempts to allocate the entire unmet CPU demand for \( s_1 \). This means that \( \min((\omega_{s_1}^{\text{req}} - \omega_{s_1}^{\text{real}}), \Omega_n^{\text{free}}) \) CPU cycles/second are allocated to application \( s_1 \). If there is not enough free memory to fit \( s_1 \), the algorithm continues with the next application \( s_2 \in S \), etc. The iteration \( k \) ends when either the free memory or CPU are exhausted, or when all the applications in \( S \) have been considered.

After each iteration \( k \), the placement controller produces a new placement solution \( R^k \) that lists a set of applications to run on the local node during the next placement cycle. At the end of the loop, the placement algorithm returns from the set \(|R| \in \{k = 0, 1, \cdots, |R|\}\) the configuration that maximizes the total number of CPU cycles/second delivered to the active applications.

Note that the algorithm takes into account the cost of stopping and starting an application, as illustrated in the following example. If, for instance, starting application \( s \) consumes the local CPU for 9 seconds and the length of the placement cycle is 900 seconds, then the cost of starting \( s \) is 1% of the total CPU resources of the node during that placement cycle. The same rule applies for stopping an application. (A more detailed model could take into account the current node utilization and the fact that the start and the stop processes do not take the entire CPU capacity.)

Phase 3: Reconfiguring and Advertising the New Configuration. The algorithm announces the new configuration for the node if the set of active applications has changed (lines 30-31). Next, it switches from the old configuration to the new configuration by stopping and starting applications (line 32). After completing these operations, the algorithm advertises its current configuration (line 33).
9.4. EXPERIMENTAL RESULTS

The placement controllers process the announcement of a new configuration, whose purpose is to reduce unnecessary configuration changes, such as several nodes activating the same idle application. The announcement of the commitment of the placement operation is used for routing requests.

Comparison with Centralized Placement Controllers

Our decentralized scheme for application placement has a number of advantages over proposed centralized solutions [72, 79, 45].

First, it scales to a large number of nodes. The controller makes decisions based on state information collected from a small subset of the nodes in the system (i.e., its neighbors), which does not grow with the system size. (As we have discussed, there is a limitation to which size the update propagation scheme scales, since it is based on maintaining a copy of the global placement matrix on each node. We are currently working on a decentralized update scheme, where each node knows about a configurable number of providers for each application.)

Second, decentralized application placement contributes to a robust system, as failures of single nodes do not affect the availability of the rest of the system. In contrast to a centralized system, all components are functionally identical.

Third, decentralized placement enables a large system to adapt to external events almost instantly, while a centralized approach does not have this capability. Consider a centralized controller that periodically gathers data about the state of the system and computes a placement solution. The timescale according to which such a system can adapt is determined by the length of the placement cycle. In the decentralized case, a large number of controllers reconfigure asynchronously and periodically. The placement operations are distributed over time and parts of the system react almost instantly to external events.

Fourth, decentralization of the application placement does not come at the cost of additional complexity. All communication takes place between pairs of nodes, without further synchronization. Furthermore, one can use the same algorithm for centralized, as well as decentralized placement.

Compared to a centralized solution, decentralized application placement also has disadvantages, as our results in the next section show. Running a controller on each node generally results in a higher processing overhead for application placement. Second, the number of changes in the sense of application starts and stops is usually larger in the decentralized case, since there is no coordination between the decisions of the local controllers.

9.4 Experimental Results

We implemented our design in Java and studied its behavior through extensive simulations. The implementation is based on the javaSimulation package [17] that provides a general framework for process-based discrete event simulation. (We also
implemented our algorithms in a real system based on the Tomcat [19] environment. Measurements from the testbed will be reported elsewhere.)

We have conducted simulations with a large number of system configurations and load patterns. We use several models for synthetic load because we are not aware of any large-scale load traces of J2EE applications. Due to space limitations, we only report results here for one type of load pattern called “power-law” [109], where applications are ranked according to their popularity and the application with rank \( \alpha \) is assigned a random weight in the interval \([0, \alpha^{-2.16}]\). To obtain the load for individual applications, these weights are normalized, and multiplied by the total CPU demand for all applications. In our experiments, the load changes periodically. The weights assigned to the applications in a given period are independent of the previous period.

Following our previous work [72], we use two parameters to control the difficulty of an application placement problem: the CPU load factor (CLF) and the memory load factor (MLF). The CLF is the ratio between the total CPU demand of the applications and the total CPU capacity available in the system:

\[
\text{CLF} = \frac{\sum_{a \in A} w_a r^c}{\sum_{n \in N} \Omega_n}.
\]

Similarly, the MLF is the ratio between the sum of the memory required by each deployed application and the total memory available in the system:

\[
\text{MLF} = \frac{\sum_{a \in A} \gamma_a r^m}{\sum_{n \in N} \Gamma_n}.
\]

In case \( \text{MLF} = 1 \), the system has enough total memory to run one instance of each application. Note that the memory is scattered across the nodes, and it might be not possible to find a placement solution for any value of \( \text{MLF} > 0 \).

We use two performance metrics to evaluate the placement algorithms: the accuracy and the number of placement changes. Accuracy is the ratio between the satisfied CPU demand and the total CPU demand. An accuracy of 1 means that the system can process all the incoming requests. The number of placement changes is the total number of application starts and stops in the entire system during one placement cycle. A good algorithm should exhibit high accuracy and a low number of placement changes.

In the simulation, the memory capacity \( \Gamma_n \) of a node is uniformly distributed over the set \{1, 2, 3, 4\} GB. The memory requirement \( \gamma_n \) of an application is uniformly distributed over the set \{0.4, 0.8, 1.2, 1.6\} GB. The CPU capacity \( \Omega_n \) is the same for all nodes and is set to 2 GHz.

Unless otherwise noted, each placement cycle lasts 900 seconds and each simulation run lasts 3600 seconds. The first cycle (900 seconds) is a warm-up period and the results from this period are discarded. The load pattern changes every 900 seconds and the performance metrics (accuracy and number of placement changes) are measured at the end of each placement cycle. During a placement cycle, nodes independently pick a random time to run the placement algorithm. The default time to start or stop an application is 0 seconds.

The measurement points in the Figs. 9.2, 9.3 are averages of 100 simulation runs.

Comparing the Decentralized and Centralized Algorithms. Fig. 9.2 shows a performance comparison between our decentralized placement controller and two cen-
Figure 9.2: Comparison between the decentralized and centralized placement algorithms for 100 nodes. Top: Placement Accuracy (left: CLF=0.9, right: MLF=0.4) Bottom: Number of changes (left: CLF=0.9, right: MLF=0.4)

tralized controllers with the same control objectives. The “base centralized” algorithm [72] (previously developed by us) uses a greedy heuristic and two network flows algorithms (max flow and min cost) on a bi-partite graph to find the optimal allocation. The “improved centralized” algorithm is our most recent, enhanced version of the “base centralized” algorithm.

The evaluation is based on a system with 100 nodes. In the decentralized case, each node has 16 neighbors. We report on two series of simulations. First, we set CLF = 0.9 and vary MLF from 0.1 to 1. For the second series, we set MLF = 0.4 and vary CLF from 0.1 to 1.

Fig. 9.2 shows the evaluation results. The accuracy of the decentralized algorithm is about 5% lower than the accuracy of the improved centralized algorithm. This cost in CPU resources is paid for a decentralized solution. We further observe that the number of placement changes in the decentralized algorithm is several
times higher than that of the improved centralized algorithm. We expect a higher number of changes in the decentralized case, because the nodes make decisions independently and do not coordinate them. We believe that the number of changes could be reduced by applying only those changes that significantly improve the control objective.

As expected, the CLF influences the accuracy significantly. For relatively low CPU utilizations (CLF \( \leq 0.5 \)), the accuracy of the decentralized algorithm is close to that of the centralized algorithm and, furthermore, close to the ideal value of 1. The decrease in accuracy accelerates for CLF \( \geq 0.8 \). In the decentralized case, the number of placement changes increases with CLF, peaks around CLF = 0.8 and decreases afterwards. We explain this decrease by the fact that, for a high CLF, many nodes are loaded close to their capacity, and it is difficult for the algorithm to find placement changes that can further improve their utilization.
9.4. EXPERIMENTAL RESULTS

![Graphs showing convergence after a load change and number of changes over simulation time.]

Figure 9.4: Convergence after a load change. Left: placement accuracy. Right: number of changes.

**Scalability of the Decentralized Placement Algorithm.** To investigate the scalability of our algorithm, we consider a system where nodes have on average 16 neighbors. We report on two series of simulations, in which we vary the number of nodes in the system from 100 to 500. First, we set $CLF = 0.9$ and vary $MLF$ from 0.1 to 1. For the second series, we set $MLF = 0.4$ and vary $CLF$ from 0.1 to 1. Fig. 9.3 shows that the accuracy tends to slightly increase by about 1-2% when the system size increases from 100 to 500 nodes. Fig. 9.3 also shows that the number of placement changes increases linearly with the size of the system.

**Convergence after a Load Change.** Fig 9.4 illustrates the system behavior during transient phases that follow changes in the external load pattern. The system has 200 nodes and each node has on average 16 neighbors. We perform three series of experiments for which the time needed to start or to stop an application is 0, 20, and 100 seconds, respectively. The length of the placement cycle is 900 seconds. The system starts in a random configuration and, at time 0, we generate a load pattern that does not change during the remainder of the simulation. We set $MLF = 0.4$ and $CLF = 0.9$.

Fig. 9.4 shows that the increase in accuracy and the number of placement changes is initially large and then levels out over time. We observe that the time $t$ needed to start and stop applications significantly impacts the convergence time of the system. Specifically, if $t$ is small (0 seconds or 20 seconds), the system converges during a single placement cycle. For large values of $t$ (100 seconds), the system needs 2-3 placement cycles to converge. When increasing $t$, the placement accuracy becomes lower, because some CPU resources are used for starting and stopping applications. As expected, we observe changes in the system even in steady state.
CHAPTER 9. A DECENTRALIZED APPLICATION PLACEMENT CONTROLLER FOR WEB APPLICATIONS

9.5 Related Work

The application placement problem, as described in Section 9.3, is a variant of the class constrained multiple-knapsack problem, which is known to be NP-hard [71]. Variations of this problem have been studied extensively in several contexts. In the area of application placement for web services, this work is closely related to the centralized placement algorithm [72] previously developed by us.

Stewart et al. [45] present a centralized method for automatic component placement that maximizes the overall system throughput. In the area of content delivery and stream processing, [75, 76, 77] describe methodologies for placing a set of operators in a network, by balancing two objectives: (a) minimizing the delay of the information delivery and (b) minimizing the bandwidth used to send the information. In the context of utility computing, [83] presents a decentralized placement algorithm that clusters application components according to the amount of data they exchange.

9.6 Conclusion

In this paper, we presented a decentralized control scheme for application placement that attempts to maximize resource utilization. Through simulations, we have shown that decentralized placement can achieve accuracy close to that of state-of-the-art centralized placement schemes (within 4% in a specific scenario). Our decentralized scheme does not include additional synchronization costs compared to a centralized solution. Moreover, conceptual advantages such as scalability, resilience and continuous adaptation to external events motivate the study of decentralized placement as an alternative to a centralized scheme.

A number of issues require further consideration. Our current design does not address the server affinity problem and the concept of the user session. In addition, we did not consider the interaction with the database tier in our design, which limits the use of the current scheme to applications that do not require transactional capabilities or state persistence. Finally, we need to improve further the scalability of our placement scheme. As we have shown, maintaining a copy of the global placement matrix at each node potentially limits scalability, as the processing load on a node increases with the system size. We are currently evaluating an adaptive update mechanism, where each node only knows about a configurable number of providers for each application.

We are in the process of implementing and evaluating the scheme described in this paper in the Tomcat environment. The placement controller runs as an application filter in Tomcat, following the approach described in [110]. We plan to evaluate the performance of our system using the TPC-W and RUBiS benchmarks.
Chapter 10

A Service Middleware that Scales in System Size and Applications

Constantin Adam\textsuperscript{1}, Rolf Stadler\textsuperscript{1}, Chunxiang Tang\textsuperscript{2}, Malgorzata Steinder\textsuperscript{2}, Michael Spreitzer\textsuperscript{2}

\textsuperscript{1} School of Electrical Engineering, Royal Institute of Technology, Stockholm, Sweden
\textsuperscript{2} IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

Abstract

We present a peer-to-peer service management middleware that dynamically allocates system resources to a large set of applications. The system achieves scalability in number of nodes (1000s or more) through three decentralized mechanisms that run on different time scales. First, overlay construction interconnects all nodes in the system for exchanging control and state information. Second, request routing directs requests to nodes that offer the corresponding applications. Third, application placement controls the set of offered applications on each node, in order to achieve efficient operation and service differentiation. The design supports a large number of applications (100s or more) through selective propagation of configuration information needed for request routing. The control load on a node increases linearly with the number of applications in the system. Service differentiation is achieved through assigning a utility to each application, which influences the application placement process. Simulation studies show that the system operates efficiently for different sizes, adapts fast to load changes and failures and effectively differentiates between different applications under overload.

10.1 Introduction

Fundamentally, our aim with this work is to develop engineering principles for large-scale autonomous systems, where individual components self-organize and

\footnotesize{This paper will be published in IM-2007. It has co-Authors from IBM Research. The role of the IBM researchers was to give comments to problem motivation and the results. The problem statement, the technical work and the writing was done by Constantin Adam.}
work together towards a common goal that we can control. In this paper, we do this in the context of application services. We present a design that dynamically allocates CPU and memory resources to a potentially large number of applications offered by a global server cluster. The novel aspect of our design is that all its functions, including resource allocation, are decentralized, which forms the basis for scalability and robustness of the system.

Our design has three features characteristic to many peer-to-peer systems. First, it scales with the number of nodes, because each node receives and processes state information only from a small subset of peers that is independent of the system size. Second, the design is robust, since all the nodes are functionally identical, and the failure of a node does not affect the availability of the rest of the system. Finally, the design enables a large system to adapt quickly to external events. As each node runs its local control mechanisms asynchronously and independently from other nodes, (periodic) local control operations are distributed over time, which lets parts of the systems re-configure shortly after events (such as load changes or node failures) occur.

Fig. 10.1 shows a possible deployment scenario for the design in this paper. The scenario includes several data centers and many entry points. The system offers a large variety of application services that are accessed through the Internet. We consider computationally intensive services, such as remote computations and services with dynamic content, e.g., e-commerce, or online tax filing. Throughout the paper, we refer to the combined set of nodes in all the data centers as the cluster.
This work builds on earlier results by us in the context of engineering scalable middleware for web services. It includes significant extensions and modifications to our earlier designs ([101], [102]), as we address important shortcomings of those designs and conduct a more thorough evaluation of the system. In [101] we introduced a design similar to the one in this paper, which is though restricted to a small number of applications, as each system application requires its own overlay. The design in this paper uses only a single overlay, which is made possible through the introduction of a forwarding table for request routing and the concept of selective propagation of state to maintain the forwarding tables. In [102] we introduced a decentralized controller for application placement. In this paper, we use a simplified version of this controller, improve its scalability, and extend it to achieve service differentiation.

The rest of the paper is organized as follows: Section 10.2 gives an overview of our P2P design. Section 10.3 evaluates the algorithm through simulation. Section 10.4 discusses related work. Finally, Section 10.5 concludes the paper.

10.2 System Design

System Model

We consider a set \( \mathcal{N} \) of nodes and a set \( \mathcal{A} \) of applications. A node can run concurrently (instances of several) applications. We assume that the CPU and the memory
are bottleneck resources on each node. The CPU is a load-dependent resource, as
its consumption depends on the offered load. The memory is a load-independent
resource, as we assume that it is consumed regardless of the offered load, i.e., even
if the program processes no requests ([72]).

We treat memory as a load-independent resource for several reasons. First,
a significant amount of memory is consumed by an application instance even if
it receives no requests. Second, memory consumption is often related to prior
application usage rather than its current load. For example, even in the presence
of a low load, memory usage may still be high because of data caching. Third, as
an accurate projection of future memory usage is difficult and many applications
cannot run when the system is out of memory, it is reasonable to be conservative
in the estimation of memory usage, i.e., by using the upper limit instead of the
average.

The Role of the Entry Point

Our design assumes the entry points have the functionality of a layer-4 switch.
While entry points with more advanced functionality, such as layer-7 switching,
load balancing, or sophisticated scheduling can provide similar functionality to
our design, or fit in our design as well, we advocate using our design together
with layer-4 switches, for the following reasons. First, the realization of a layer-4
entry point is generally simpler, more efficient and more economical than that of
a layer-7 switch [23]. Second, our approach is potentially more resilient. Since a
layer-4 entry point does not hold any state information about load and resource
usage, its failure will affect only the pending requests. Third, our solution scales
better. Policies such as service differentiation and quality of service objectives are
realized through the application placement mechanism, which runs on every node
of the system, rather than on the entry point. A single entry point can manage
resources for only a limited number of nodes. Increasing the number of entry points
makes synchronization among them necessary and adds to the system complexity.
Fundamentally speaking, supporting sophisticated resource allocation strategies on
an entry point has all the disadvantages inherent to centralized control schemes,
when compared to the decentralized scheme we advocate.

Decentralized Control Mechanisms

Three distributed mechanisms form the core of our design, as shown in Fig. 10.2.
Topology construction uses an epidemic protocol and a set of local rules to organ-
ize dynamically the nodes into an overlay, through which they disseminate state
and control information in a scalable and robust manner. Request routing directs
service requests towards available resources. This mechanism maintains the for-
warding tables of the nodes by disseminating information about which nodes run
which applications. Application placement dynamically assigns the cluster resources
(CPU and memory) to applications. Resource allocation is achieved by continu-
ously maximizing a utility function that takes into account the external load, the operational states of the nodes, and the management policies. All three mechanisms run independently and asynchronously on each node.

**Overlay Construction**

Nodes self-organize into a logical overlay network in which the links are application-level (TCP) connections. Nodes use the overlay to disseminate routing information and retrieve states/statistics from their neighbors.

For the purpose of our architecture, we searched for algorithms that produce a bi-directional overlay graph with a constant degree. The topological properties of such an overlay facilitate balancing the control load. They also simplify unbiased estimation of the system state from a subset of nodes, as the state of each node can be sampled the same number of times by other nodes and each node has a neighborhood of the same size for retrieving state information. In addition, we wanted to find algorithms that allow the overlay to regenerate quickly after failures and to optimize it dynamically according to specific criteria.

Our overlay construction mechanism includes two protocols that run independently of each other. First, an epidemic protocol (CYCLON [15]) maintains on each node an up-to-date cache of active nodes that can be chosen as neighbors. Second, a protocol (GoCast [16]) maintains the list of neighbors of a node and picks new neighbors from the cache as needed. (In our implementation, each node has four neighbors, while the size of the cache is 10.)

CYCLON maintains on each node a cache of size \( c \). Each cache entry has the following format: \(<\text{address}, \text{timestamp}, \text{state information}\>\). CYCLON periodically picks the cache entry with the oldest timestamp, and sends the content of the local cache to the node with that address. The remote node replies by sending a copy of its own cache. Both nodes then merge the two caches, sort the merged list according to the timestamps and keep the \( c \) most recent entries. It has been shown in [15] that the caches produced by CYCLON represent adjacency lists of a graph with random-graph properties.

The GoCast protocol [16] selects the neighbors of the local node from the entries of the CYCLON cache. GoCast ensures that the overlay converges to a graph where each node has approximately the same number of neighbors \((\pm 1)\), which is configurable. In addition, the protocol can continuously adjust the overlay graph following a specific optimization objective, such as finding overlay neighbors with lowest latency ([16]).

CYCLON and GoCast, in combination, achieve our design goals of fast regeneration of the overlay after failures and the possibility of dynamically optimizing the overlay topology according to specific criteria. In case of a node failure, a node can replace a failed neighbor with an active node from its cache. By examining the content of the \textit{timestamp} and \textit{state information} fields of the entries in its cache, a node can identify a set of candidate neighbors that improve the optimization criteria for the overlay.
CHAPTER 10. A SERVICE MIDDLEWARE THAT SCALES IN SYSTEM SIZE AND APPLICATIONS

Table 10.1: Sample forwarding table of a node with length 3.

<table>
<thead>
<tr>
<th>App ID</th>
<th>Nodes Offering Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td>10.10.1.9 10.10.1.72 10.10.2.33</td>
</tr>
<tr>
<td>App 2</td>
<td>10.10.1.99</td>
</tr>
<tr>
<td>App 3</td>
<td>10.10.1.199 10.10.3.45 10.10.1.79</td>
</tr>
<tr>
<td>...</td>
<td>... ... ... ...</td>
</tr>
<tr>
<td>App m</td>
<td>10.10.1.232 10.10.2.40</td>
</tr>
</tbody>
</table>

The entry points are connected to the overlay in the following way. Each entry point runs an instance of the CYCLON protocol with a very large cache. It does not run GoCast, however. This ensures that each entry point is connected to a large number of active nodes, which facilitates balancing the load related to request routing. Also, the entry point does not appear as the neighbor of any node in the overlay.

Request Routing

Service requests enter the cluster through the entry points (see Fig. 10.1), which function as layer-4 switches. Each entry point is connected via logical links to a large number of nodes. An entry point directs incoming requests to these nodes using a forwarding policy. In our design, we use a round-robin forwarding policy.

Upon receiving a request from an entry point, a node determines its application type. The node processes the request if it runs the corresponding application and if its CPU utilization is below a configurable threshold $cpu_{max}$. Otherwise, the node consults its forwarding table and routes the request to a peer that offers the required application. In the absence of overload, a request will likely be routed once inside the cluster, since we believe that a single node will typically offer a small fraction of the applications in the cluster.

Table 10.1 shows an example of a node’s forwarding table. It contains, for each application offered in the system, a configurable number of nodes that run that particular application. In Table 10.1 this number is 3. The forwarding tables are maintained through the mechanism discussed in Section 10.2.

In order to avoid forwarding loops, a request header contains the addresses of the nodes that have routed the request. In addition, the number of times that a request can be forwarded is limited (to four in our design). The system drops the requests that exceed this limit.

Selective Propagation of the Routing Updates

As discussed in Section 10.2, a node periodically computes the list of applications that it will be running during the next placement cycle. After each such computa-
Table 10.2: Performance comparison between the schemes for disseminating configuration updates.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>num msgs/node</th>
<th>message length</th>
<th>load per node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flooding</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Global Placement Matrix</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Selective Propagation</td>
<td>O(A)</td>
<td>O(1)</td>
<td>O(A)</td>
</tr>
</tbody>
</table>

tion, the node advertises the list of applications to the other nodes, which update their forwarding tables. In this subsection, we propose a scheme for dynamically updating the forwarding tables in an efficient and scalable manner.

A straightforward solution to updating the forwarding tables is for each node to flood periodically its application list across the overlay. (This flooding scheme is similar to the propagation of link weight updates in OSPF.) Given the fact that each node generates one update per placement cycle and that the number of neighbors of a node is bounded by the connectivity of the overlay, a node processes $O(N)$ update messages during each placement cycle, where $N$ is the system size. Since the size of the application list is bounded, the length of an update message is bounded as well, and the processing load on a node required to update the forwarding table grows with $O(N)$.

In [102], we proposed an update scheme where the forwarding table on each node contains the list of applications that are running on every other node in the system. This table is called the global placement matrix. Maintaining a copy of the matrix at each node limits the scalability of the system, as the number of update messages increases linearly with the system size. The scheme enforces a maximum message rate on each overlay link, by buffering all the updates received by a node during a time interval $\tau$ and aggregating them into a single message. Therefore, a node will process at most $c/\tau$ update messages per second, $c$ being the connectivity of the overlay. Note, however, that the size of an update message increases with $O(N)$, and therefore the processing load on a node increases linearly with $N$.

In this paper, we introduce an approach that involves selective update propagation to achieve a processing load on a node that is independent of the system size. The basic idea is that a node propagates further only those updates that cause a modification to its forwarding table.

Upon receiving an update message, a node modifies its forwarding table as follows. It extracts from the update four variables: the sender-id containing the address of the original sender node of the update, the timestamp indicating when the update was created by the sender, the application-list of the sender and the distance in overlay hops between the sender and the current node. For each application app in the list, the node performs the following operations. First, it searches for app in its forwarding table. If app is not found, the node adds the tu-
CHAPTER 10. A SERVICE MIDDLEWARE THAT SCALES IN SYSTEM SIZE AND APPLICATIONS

ple \(<app,<sender-id,timestamp,distance>>\) to its table. (For reasons of simplicity, the sample forwarding table in Table 10.1 has a different format and does not include timestamp and distance.) Otherwise, if \(app\) is in the table, the node searches for an entry of the form \(<app,<sender-id,*,*>>\). If such an entry exists, but it is less recent than the update, then its timestamp is updated. In case there is no entry of the form \(<app,<sender-id,*,*>>\) in the forwarding table, the node adds the tuple \(<app,<sender-id,timestamp,distance>>\) to its table, if there is space in the table. In case there is no space, the node overwrites an existing tuple if the update has a smaller distance to the sender.

The update message to be further propagated by the node is constructed as follows. The node removes from the application-list of the received message each application that does not lead to a modification of its forwarding table. If the application list is empty, the node does not propagate the update any further. Otherwise, it sends the update to its neighbors, with the exception of the node where the original update came from.

For robustness purposes, the entries of the forwarding table are soft state, and each entry expires after a preconfigured timeout. For this reason, each node advertises its complete configuration after each placement cycle.

Regarding the performance of the selective update propagation, we note that the load on each node depends on the connectivity of the overlay \(c\) and on the number of applications \(A\), but is independent of the system size \(N\). A node processes \(O(A)\) update messages during a placement cycle. Since the size of the application list is bounded, the length of an update message is bounded as well, and the processing load on a node required to update the forwarding table grows with \(O(A)\).

Table 10.2 compares the performance of the three schemes for propagating updates. The columns indicate the number of update messages that each node processes per placement cycle, the length of these update messages, and the processing load induced on a node by the update propagation, computed as the product of the first two columns.

While in the flooding scheme and the global placement matrix the processing load increases with the system size, in the selective update propagation scheme the load increases with the number of applications, but is independent of the system size. We can therefore conclude that the selective propagation scheme scales with the system size.

Note that for certain parameter ranges of the system size and the number of applications, flooding might perform similar to the selective update scheme. As our simulations show, this happens when a system that has a small size runs a large number of applications (see Fig. 10.6).

An interesting property of the selective dissemination scheme is that it distributes the nodes offering an application uniformly in the forwarding tables of the other nodes. We will study this property in more detail in future work.
10.2 SYSTEM DESIGN

Application Placement

The goal of application placement is to maximize continuously a utility function. The optimization process takes into account, for each application, the external demand for resources, the supply provided by each node and a utility parameter that captures the relative importance of each application. This approach allows the system to adjust dynamically its configuration to changes in the external load, or to a new management policy that changes the relative importance of an application. Our approach to maximizing a global utility function is to perform a local optimization on each node, based on state information from its neighborhood. Our earlier work ([101, 111]) shows that such a heuristic can produce a solution that is close to optimal.

We model the problem as follows. For a node $n \in \mathcal{N}$, let $\Gamma_n$ and $\Omega_n$ be its memory and CPU capacities, respectively. Let $\mathcal{A}$ be the set of applications offered by the system and $\mathcal{R}_n$ the set of applications that run on node $n$. (A node generally runs a small subset of all the applications offered by the system.)

For an application $a \in \mathcal{A}$, let $\gamma_a$ be the memory demand of an instance of $a$ and $\omega_n^{demand}$ the total CPU demand for $a$ in the system. Let $\omega_n^{supply}$ be the CPU demand on node $n$ for $a$ and $\omega_n^{supply}$ the CPU supply that $n$ allocates to $a$. Each application $a$ has a utility parameter $u_a$ that defines its relative importance.

(The CPU demand and the CPU supply are measured in CPU cycles/second. We assume that all the nodes have the same CPU architecture and therefore an application request needs the same number of CPU cycles on all the nodes.)

We define the utility provided by a node $n$ as the weighted sum of the CPU resources supplied to each application: $utility_n = \sum_{a \in \mathcal{R}_n} \omega_n^{supply} u_a$, and the system utility as: $Utility = \sum_{n \in \mathcal{N}} \sum_{a \in \mathcal{A}} \omega_n^{supply} u_a$.

We state the problem of application placement as follows:

$$\max \sum_{n \in \mathcal{N}} \sum_{a \in \mathcal{A}} \omega_n^{supply} u_a$$

such that

$$\forall n \in \mathcal{N} \quad \Gamma_n \geq \sum_{a \in \mathcal{R}_n} \gamma_a$$

$$\forall n \in \mathcal{N} \quad \Omega_n \geq \sum_{a \in \mathcal{R}_n} \omega_n^{supply}$$

Formulas 10.2 and 10.3 stipulate that the allocated CPU and memory resources on each node cannot exceed the node’s CPU and memory capacities, respectively.

Our placement algorithm executes periodically on each node. The time between two executions of the algorithm is called the placement cycle.

The placement algorithm has three consecutive phases. In the first phase, a node gathers placement and load information locally, as well as from its neighbors. In the second phase, the node determines a set of applications to run locally during the
CHAPTER 10. A SERVICE MIDDLEWARE THAT SCALES IN SYSTEM SIZE AND APPLICATIONS

1. class AppInfo {
2.     string app_id;
3.     double cpu_demand, cpu_supply, utility_parameter;
4. }
5. List<AppInfo> active_apps, standby_apps, new_active_apps, all_apps;
6. double max_utility, utility;
7. 
8. while(true) {
9.     active_apps=getLocalActiveApps();
10.    all_apps=getApps(forwarding_table);
11.    standby_apps=all_apps-active_apps;
12.    neighbors=getOverlayNeighbors();
13.    standby_apps=getAppStats(neighbors, standby_apps);
14.    active_apps=sortDecreasing(active_apps);
15.    standby_apps=sortDecreasing(standby_apps);
16.    new_active_apps=active_apps;
17.    max_utility=currentUtility();
18.    for(i=0..active_apps.size()) {
19.        utility=transferResources(top i active_apps, standby_apps);
20.        if(utility>(max_utility+change_cost)) {
21.            max_utility=utility;
22.            new_active_apps=active_apps-top_i_active_apps+sel_standby_apps;
23.        }
24.    }
25.    advertise(new_active_apps);
26.    if(new_active_apps!=active_apps)
27.        stopAndStartApps(active_apps, new_active_apps);
28.    wait end of placement cycle;
29. }

Figure 10.3: The pseudo-code of the application placement mechanism.

next placement cycle. In the last phase, the node carries out the placement changes (i.e., start or stop of applications) and advertises its new placement configuration to other nodes. We give the pseudo-code of the algorithm in Fig. 10.3 and describe each of its phases below.

Phase 1: Gathering State Information. A node retrieves the set of active applications $R$ that it is currently offering and it gathers statistics (CPU supply and demand) for each application in $R$ (line 8). A node also retrieves from its forwarding table a list of all the applications offered in the system (line 9). Any application that is offered by the system, but is not currently active on the node becomes a potential candidate to be activated on the node during the current
10.2. **SYSTEM DESIGN**

placement cycle and is added to a standby set \( S \) (line 10). A node retrieves from each overlay neighbor \( x \) the list of applications \((a_1 \cdots a_m)\) running on \( x \), the memory requirements \((\gamma_{a_1} \cdots \gamma_{a_m})\) of those applications, the CPU cycles/second \((\omega_{x,a_1}^{\text{supply}}, \cdots, \omega_{x,a_m}^{\text{supply}})\) delivered to those applications, and the CPU demands of those applications \((\omega_{x,a_1}^{\text{demand}}, \cdots, \omega_{x,a_m}^{\text{demand}})\). In addition, neighbor \( x \) also reports the locally measured demands for applications it could not route, since they are not offered in the system (lines 11-12). (A high demand or utility for these inactive applications might trigger their activation during the next placement cycle.)

**Phase 2: Computing a New Set of Active Applications.** The placement algorithm attempts to replace a subset of applications in \( \mathcal{R} \) with a subset of applications in \( S \), so that the utility of the node is maximized. Since the candidate space for the optimal configuration grows exponentially with \(|\mathcal{R} \cup S|\), we apply a heuristic that reduces the complexity of the problem to \( O(|\mathcal{R}| + |S|) \) (lines 13-23).

On a node \( n \), we remove from \( S \) all the applications for which the supply matches the demand. The active applications in \( \mathcal{R}_n \) are sorted in increasing order of their utility which we define as the lead delivered by \( n \) to application \( a \), multiplied by the utility coefficient of \( a \), i.e., \( u_a \omega_{n,a}^{\text{supply}} \). The applications in the standby set \( S \) are sorted in decreasing order of the amount that the node would add to its utility in the case when it would fulfill the unsatisfied demand for those applications, i.e., \( u_a \sum_{n \in \text{neighbors}} (\omega_{n,a}^{\text{demand}} - \omega_{n,a}^{\text{supply}}) \). Intuitively, the placement controller tries to replace low-utility applications from \( \mathcal{R} \) with high-utility applications in \( S \), so that the local utility is maximized.

The placement algorithm has \(|\mathcal{R}| + 1\) iterations \((k = 0 \cdots |\mathcal{R}|)\). During the first iteration \((k = 0)\), it does not remove any application from \( \mathcal{R} \). If the local node \( n \) has available memory and CPU cycles (i.e., \( \Gamma_{h}^{\text{free}} > 0 \) and \( \Omega_{h}^{\text{free}} > 0 \)), then the algorithm attempts to add one or more applications from \( S \) to \( \mathcal{R} \). This is done by selecting applications from the top of \( S \), subtracting the cost for starting the applications, and evaluating the resulting gain in utility.

During iteration \( k > 0 \), the algorithm removes the top \( k \) applications from \( \mathcal{R} \). It then computes the available memory and CPU resources and attempts to assign these resources to applications in \( \mathcal{S} \) in the following way. The algorithm attempts to fit the first application \( S_1 \in \mathcal{S} \) into the available memory. If this operation succeeds, then the algorithm attempts to allocate the entire unmet CPU demand for \( S_1 \). This means that \( \min(\omega_{s_1}^{\text{req}} - \omega_{s_1}^{\text{cal}}, \Omega_{h}^{\text{free}}) \) CPU cycles/second are allocated to application \( S_1 \). If there is not enough free memory to fit \( S_1 \), the algorithm continues with the next application \( S_2 \in \mathcal{S} \), etc. The iteration \( k \) ends when either the free memory or CPU are exhausted, or when all the applications in \( \mathcal{S} \) have been considered.

After each iteration \( k \), the placement controller produces a new placement solution \( \mathcal{R}^k \) that lists a set of applications to run on the local node during the next placement cycle. At the end of the loop, the placement algorithm returns from the set \( \{\mathcal{R}^k|k = 0, 1, \cdots, |\mathcal{R}|\} \) the configuration that maximizes the utility of the node.

The algorithm computes the cost of stopping and starting applications (the
CHAPTER 10. A SERVICE MIDDLEWARE THAT SCALES IN SYSTEM SIZE AND APPLICATIONS

change_cost variable, defined in line 19) by multiplying the current utility of the node by the time needed to start and stop the applications computed in this phase. In this way, a node that delivers a low utility is more likely to change its configuration.

Phase 3: Reconfiguring and Advertising the New Configuration. The algorithm advertises its current configuration (line 24) and switches from the old configuration to the new configuration by stopping and starting applications (line 25).

10.3 System Evaluation through Simulation

Evaluation Setup

We evaluate our system through simulation according to five criteria. Efficiency captures the capability of the system to operate exhibiting high performance parameters in a steady state. Scalability captures the capability of the system to maintain similar performance characteristics when its size increases. Adaptability captures the capability of the system to respond to a change in the operating conditions by reconfiguring and converging to a new steady state. Robustness captures the capability of the system to respond to node arrivals, departures and failures, by reconfiguring and converging to a new steady state. In this paper, we understand manageability as the capability of the system to adjust its configuration to changes in management policies.

The simulated system is written in Java and uses javaSimulation [17], a package that provides a general framework for process-based discrete event simulation. (We also implemented our design on a testbed using Tomcat [19]. Measurements and experiences from the testbed will be reported elsewhere.)

We simulate a node as having two FIFO message buffers that store the service requests and the control messages, respectively. Every 10 ms the node processes all messages in these two buffers. Therefore, each message experiences a delay between 0 and 10 ms, before being processed. In the case of the request buffer, the node moves a request that it can serve locally into its list of active requests being executed. This list has a capacity of 50, which means that a node can concurrently execute 50 service requests. As a request has an average execution time of 250 ms (see below), the service capacity of a node is 200 req/sec. If a service request cannot be executed locally, it is forwarded to another node or dropped. The memory capacity $\Gamma_n$ of a node is uniformly distributed over the set \{1, 2, 3, 4\} GB.

For the simulations, the size of the cluster varies between 50 and 800 nodes, and there is one entry point per 200 nodes. If a system has multiple entry points, the request generator distributes the external load equally among them. In all experiments, we set the cache size for CYCLON to 10 and the target number of overlay neighbors for GoCast to 4. A node gathers state information from all the nodes within two overlay hops (a maximum of 17 nodes for a network where each node has 4 neighbors). Each node runs CYCLON every 5 sec, GoCast every 1 sec.
and application placement every 30 sec. A node limits the size of its forwarding table to four nodes for each application, unless otherwise specified.

We assume that a request for each application has the same average execution time of 250 ms. On a node, the execution times of the requests follow an exponential distribution with $\lambda = 4 \text{ sec}^{-1}$. We assume further that the memory requirement $\gamma_a$ for an instance of application $a$ is uniformly distributed over the set {0.4, 0.8, 1.2, 1.6} GB.

We have conducted simulations with a load pattern based on the power-law distribution where applications are ranked according to their popularity and the application with rank $a$ is assigned a value proportional to $a^{-2.16}$ [109]. We obtain the average external load (measured in CPU cycles/second) for the application with rank $a = 1, 2, 3 \ldots$ by choosing a random weight in the interval $[0, a^{-2.16}]$, normalizing the resulting distribution, and multiplying the distribution with the total external load. The arrival of individual service requests is modeled as a Poisson process.

Each simulation run lasts 600 sec. We start measuring after 210 sec, which represents the warm-up phase for the simulation. Each point in Figs. 10.4-10.9 represents the average of 20 simulation runs.

For all experiments except manageability, we assign the same utility parameter to all applications, which means that the global utility function that the system attempts to maximize is the utilization of its combined CPU resources.

In all experiments, we use three output metrics to measure the system performance. Satisfied Demand is the ratio between the CPU resources supplied by the system to the applications and the total CPU demand (which we also call the external load). Since we assume that a request for each application takes, on average, the same amount of CPU resources, the satisfied demand can be understood as the rate of the executed requests divided by the rate of the incoming requests, i.e., the ratio between service rate and arrival rate. A satisfied demand of 1, for instance, means that the system processes all the incoming requests. One minus the satisfied demand gives the fraction of the requests that are dropped by the system. The number of configuration changes is the (average) number of application-start and application-stop operations that a node performs during one placement cycle. The control load gives the (average) number of control messages received by a node in one second. The messages are generated by the control mechanisms of the system, namely, overlay maintenance, dissemination of the routing updates, and retrieval of the neighborhood state.

Evaluation Scenarios

System Efficiency

We assess the system efficiency for various intensities of the external load. The load intensity is represented using the CPU load factor (CLF), as defined in [72]. The
CLF is the ratio between the total CPU demand of the applications and the total CPU capacity available in the system: \( CLF = \sum_{a \in A} \omega_a^{\text{demand}} / \sum_{n \in N} \Omega_n. \)

Fig 10.4 shows the system performance for different values of CLF up to 2.0, which corresponds to an arrival rate that is double of the maximum service rate. The system under evaluation has 200 nodes and runs 100 applications.

As expected, the satisfied demand decreases and the number of configuration changes increases when the external load increases. The system utilization, computed as the product of satisfied demand and CLF, increases almost linearly until a CLF of 0.6, then flattens out and becomes almost constant for CLF > 1.2.

The maximum (average) utilization that our system achieves under these conditions is about 95%. The main limiting factor of achieving higher utilization stems from memory fragmentation. According to our simulation configuration, some applications, which require a large amount of memory (1.2GB or 1.6GB), cannot run on certain nodes (that have only 1GB of memory). This reduces the number of nodes that can run these applications. While an ideal system (i.e., a centralized system that has the combined resources of all the nodes in the simulated system)
can generally achieve a utilization of 100\%, memory fragmentation limits in a similar way the performance of a centralized resource controller that manages all the system nodes.

An unexpected result is that the control load per node decreases with the external load. The fraction of the control load that changes with the external load stems from routing updates (see Section 10.2). We explain the measurement result by the way in which the placement algorithm chooses the set of active applications on a node. Since it sorts the applications in the order of their decreasing unsatisfied demand, it favors high-demand applications in its decisions. Consequently, low-demand applications tend to be stopped and, therefore, the number of active applications in the system tends to decrease as the external load increases. The decrease in active applications means a smaller number of routing updates, and, therefore, a lower control load. To avoid starvation of the low-demand applications, a user can increase the utility parameters for these applications, as described in Section 10.3.
Figure 10.6: System scalability. The x-axis represents the number of nodes in the system.

Impact of the Forwarding Table Size on the System Efficiency

In this set of experiments, we measure the system performance while varying the size of the forwarding table. These measurements allow us to find a tradeoff between the control load for update dissemination on the one hand, and satisfied demand and configuration changes on the other.

We run the experiments for a system with 200 nodes. Fig. 10.5 gives the results for the case where the length of each row in the forwarding table is 1, 2, 4, 8, 16 and 200.

We observe an initial significant gain in satisfied demand, from a length of 1 to a length of 4, above which the gain in satisfied demand becomes very small. As expected, the number of configuration changes decreases with the length, while the control load increases with the length. These experiments led us to choose 4 as the length of the rows in the forwarding table.
10.3. **SYSTEM EVALUATION THROUGH SIMULATION**

**System Scalability**

To assess the system scalability, we vary the system size from 50 to 800 nodes, while the number of applications is set to 100. Fig. 10.6 shows the results of these experiments.

We observe that the satisfied demand and the (average) number of configuration changes per node do not depend on the size of the system, within the parameter range we are measuring. This shows that request routing effectively balances the load for all system sizes considered.

The control load per node increases up to a system size of 200 nodes, then flattens and slightly decreases. We explain the initial increase in load by the fact that our approach, selective update dissemination, performs as well as flooding, as long as the system size is comparable to the number of applications (see Section 10.2). When the system size increases above 200 nodes, our approach reduces the load compared to flooding. As we argued in Section 10.2, for a large system size, the control load does not depend on the number of nodes, but only on the number of applications. This leads us to the conjecture that this system can scale to any size regarding the performance metrics considered here, as long as the number of applications remains bounded. This is a useful feature for systems with a large number of machines and only a limited number of applications, such as online auction sites.

**System Adaptability**

We evaluate the system adaptability for the case where new applications and additional load are added to a system with 400 nodes. Initially, the system is in steady state, offering 50 applications and having an external load of CLF=0.4. After 300 sec, we add instantly 50 new applications that induce an additional load of CLF=0.4 in the system, which brings the total load to CLF=0.8. Fig 10.7 shows the result of this experiment.

As expected from Fig. 10.4, after adding new load to the system, the satisfied demand decreases, the number of configuration changes increases and the load per node decreases. After the system reaches a steady state, the performance metrics show values that are consistent with the measurements in Fig. 10.4.

Surprisingly for us, we observe that all three output metrics show different settling times. The satisfied demand converges in about two placement cycles, the number of configuration changes in four, and the control load on a node in about seven. We explain this divergence by the imperfection in the decentralized placement, where nodes continue making changes to their local configuration without improving the satisfied demand.

**System Robustness**

We evaluate the system robustness for the case where a subset of the nodes in the system fails. The initial size of the system is 400 nodes and the external load is CLF=0.5. We run two experiments in which a system in steady state experiences
node failures after 300 sec. In the first experiment, 10% of the nodes fail. As a result, the amount of system resources decreases and the load becomes CLF=0.55. In the second experiment, 50% of the nodes fail and the load thus becomes CLF=1.0. Fig. 10.8 shows the system reaction for both experiments.

After the failures, the system reaches a new steady state. We observe that, if 10% of the nodes fail, the output metrics are approximately the same as before the failures. In the case where 50% of the nodes fail, the output metrics change significantly. We explain the difference in the outcome of these two experiments with the fact that, in the first case, the load increases from CLF=0.5 to 0.55, while, in the second case, the load increases from CLF=0.5 to 1.0 (which means overload). Note that these output metrics are consistent with the values in Figs. 10.4 and 10.6.

The system recovers surprisingly fast and reaches a steady state, even after massive failures. We explain this behavior by the fact that the overlay construction mechanisms operate on a faster timescale than the application placement (1 sec for Gocast and 5 sec for CYCLON, vs. 30 sec for the application placement). Therefore, with very high probability, the overlay is rebuilt completely within one
placement cycle after a failure. The rebuilt overlay enables all nodes to function properly again, specifically, to receive routing updates and state information from their neighborhood. As the overlay recovers fast, the system reacts to a failure similarly as it would to a sudden increase in load.

As in the case of Fig. 10.7, all the three output metrics have different convergence times, and the satisfied demand converges first, the number of configuration changes second, and the control load third.

**System Manageability**

The management policy in this scenario enables the system administrator to provide differentiated service by changing, at runtime, the utility parameter for each application (see Section 10.2). Increasing the utility parameter of a specific application results in an increased satisfied demand for that application and its deployment on a larger number of nodes. (We define the satisfied demand of an application as the ratio between the rate of serviced requests divided by the request arrival rate for that application.)
CHAPTER 10. A SERVICE MIDDLEWARE THAT SCALES IN SYSTEM SIZE AND APPLICATIONS

Figure 10.9: Service differentiation. The x-axis represents the simulation time (in seconds).

For this experiment, a system with 400 nodes offers 50 applications, and the external load is CLF=1. Initially, all the applications have the same utility parameter, equal to 1. After 300 sec, the utility parameter of the applications 3 and 4 is increased from 1 to 10.

Fig. 10.9 illustrates how the system reacts to this change. We show the satisfied demand for the applications 1, 2, 3 and 4 (out of all 50 applications). Applications 1 and 2 do not experience an increase in utility, while applications 3 and 4 do. The figure shows how the changes of the utility parameters affect the satisfied demand and the number of nodes on which each application is active.

We observe that the change in the utility parameters significantly increases the satisfied demand for applications 3 and 4, while the satisfied demand for applications 1 and 2 decreases. (The average satisfied demand in the system is around 87%). At the same time, the number of nodes on which applications 3 and 4 are active increases, while the number of nodes on which applications 1 and 2 are active decreases.

We conclude that our system can effectively provide service differentiation in an overload scenario, and that the system administrator can control the service differentiation through the utility parameters. Increasing the utility parameter for an application usually does not only increase its satisfied demand, but also its resilience to failures, since the number of nodes on which it is active is increased.

Note that predicting the satisfied demand for a given utility parameter is difficult to achieve, since it depends on the demand for all the applications in the system. The current design allows increasing the satisfied demand for a specific application only up to a certain point that depends on the external load. To achieve a 100% satisfied demand for a specific application, changes to the scheduling mechanism on the nodes are needed.
10.4 Related Work

The application placement problem, as described in Section 10.2, is a variant of the class constrained multiple-knapsack problem, which is known to be NP-hard [71]. Variations of this problem have been studied extensively in several contexts. In the area of application placement for web services, this work is closely related to the centralized placement algorithm [72].

Stewart et al. [45] present a centralized method for automatic component placement that maximizes the overall system throughput. In the area of content delivery and stream processing, [75, 76, 77] describe methodologies for placing a set of operators in a network, by balancing two objectives: (a) minimizing the delay of the information delivery and (b) minimizing the bandwidth used to send the information. In the context of utility computing, [83] presents a decentralized placement algorithm that clusters application components according to the amount of data they exchange.

10.5 Discussion and Future Work

In this paper, we have presented a peer-to-peer design for a self-organizing middleware in support of application services. Our design manages several types of system resources, including CPU and memory. It scales in terms of nodes and supports a large number of applications. We use three decentralized control mechanisms in our design: overlay construction, request routing and application placement, each of which runs on a different timescale to achieve its own objective.

We highlight here two key results derived from the evaluation of the design through simulation. First, we have argued that the selective propagation of routing updates introduced in this paper allows the system to scale to any size. This is because the message load on a node, induced by all control mechanisms, does not increase, as long as the number of applications remains bounded. We have verified through simulation that a system can scale from 200 to 800 nodes without performance degradation in satisfied demand, number of configuration changes and load per node.

Second, service differentiation in our system can be controlled effectively through changing the values of the utility parameters on the application placement controller. Increasing the utility parameter for a specific application increases the satisfied demand for that application, which means that fewer requests for it are dropped. Further, the resilience of the application to node failures increases, as more nodes start offering it. We have demonstrated service differentiation under overload conditions (see Fig. 10.9).

We are in the process of implementing and evaluating the design described in this paper in the Tomcat environment. The request routing mechanism runs as an application filter in Tomcat, following the approach we described in [110].
CHAPTER 10. A SERVICE MIDDLEWARE THAT SCALES IN SYSTEM SIZE AND APPLICATIONS

We are evaluating the performance of the system using the TPC-W and RUBiS benchmarks.

A number of issues require further consideration. Our current design does not address the server affinity problem and the concept of the user session. In addition, we did not consider the interaction with the database tier in our design, which limits the use of the current scheme to applications that do not require transactional capabilities or state persistence.

10.6 Acknowledgments

This work was supported by the Graduate School of Telecommunications (GST) at KTH and an IBM Faculty Award.
Chapter 11

Implementation of a Service Middleware that Scales in System Size and Applications

Constantin Adam
Laboratory for Communication Networks
KTH Royal Institute of Technology,
Stockholm, Sweden

Abstract

This report describes the implementation of the design described in Chapter 10 on a testbed with 14 application servers. The prototype is evaluated using the RUBiS benchmark. The results show that the testbed implementation behaves similarly to the simulation, therefore validating our simulation model and proving the feasibility of the design. Specifically, the results show that the prototype, within the parameter range tested, (a) operates efficiently under (light) overload, (b) quickly adapts to the addition of applications and corresponding load and (c) allows for effective service differentiation by a system administrator.

11.1 Implementation

Integrating Chameleon into the Tomcat Framework

We have implemented our middleware, which we call Chameleon, in Java. The middleware runs as an application filter in the Tomcat environment [19] (Fig. 11.1). The filter is initialized when the Tomcat server starts the web application(s) associated with it. During the initialization phase, a number of threads are started that run the three decentralized control mechanisms (Overlay Construction, Request Routing and Application Placement), as well as the Load Calculator.
As shown in Fig. 11.1, the Tomcat connector forwards incoming requests to the routing mechanism that is running inside the filter. This mechanism decides whether the node should process the request locally, forward it to a peer, or drop it. That decision is reached as follows. Routing identifies the application needed to process the request and checks whether it is locally available. If the application runs
locally and the node is not overloaded, the request is sent to the application servlet for local processing, as shown in Fig. 11.1(a). Otherwise (as shown in Fig. 11.1(b)), routing reads (from the ‘pragma’ field of the HTTP header) the path of the nodes visited by the request since it entered the cluster. If the length of the path is smaller than a preconfigured value (in our current configuration 2), the routing mechanism forwards the request to a node that can process it and that has not been visited by the request. If the request has already been forwarded too many times or cannot be routed, the node drops it and sends the client an error page.

In order to determine whether a node is overloaded, the Load Calculator measures the CPU utilization on that node and counts the number of threads that are currently processing a request, or are waiting for a reply to a forwarded request. A node is overloaded, if either: (a) its CPU utilization is above a certain threshold (85% in our current configuration), or (b) if the number of processing threads exceeds specific thresholds (in our current configuration, the thresholds are 10 for processing threads and 80 for forwarding threads).

The Load Calculator maintains the thread statistics mentioned above, because the request arrival process can be very bursty and the service time for requests can vary significantly. Under these conditions, periodic measurements of the CPU utilization alone are not sufficient to determine the current state of a node. As an example, assume the Load Calculator samples the CPU utilization every second, and averages over the last five measurements. If no requests arrive during the first four seconds (and the CPU utilization thus is 0%) and the node receives a burst of requests during the last second (driving its CPU utilization up to 100%), then the average CPU utilization would be calculated as 20% (which would be below the overload threshold of 85%). However, the node is in reality overloaded, and the estimation of the current state is therefore wrong. Reducing the measurement cycle for the CPU utilization would not solve this problem, because the situation described above can occur on a faster timescale. Maintaining thread statistics (in addition to estimating the CPU utilization) helps eliminate this problem.

We have implemented a two-way approach to forward requests, in which the entry points and the application servers act as TCP gateways ([104], [26]). This technique, also known as “request chaining”, works as follows. The client establishes a TCP connection with an entry point. When the entry point or a node forwards a request to a peer, it opens a TCP connection and sends the request over this connection. The reply travels back from the node that handles the request to the client following the TCP connections opened during the request forwarding phase.

11.2 Evaluation

Testbed and Setup

The test bed that we have set up for the evaluation includes ten low-capacity nodes (with Celeron 1.7 GHz processors and 128 MB RAM), four high-capacity nodes (with P4 2 GHz processors and 256 MB RAM) and four entry points (with
Figure 11.2: The configuration of the experimental platform. In the text, the application servers are referred to as nodes.

P4 2 GHz processors and 256 MB RAM, all connected via 100Mbps Ethernet Switches, as shown in Fig. 11.2. We use four additional machines as client emulators for running the RUBiS benchmark, as described below.

The RUBiS benchmark

We evaluate our system using RUBiS, a benchmark modeled after eBay.com that implements the core functionality of an auction site: selling, browsing and bidding. Because our evaluation focuses on the application servers (not on the database servers), we evaluate the prototype in the browsing mode of RUBiS, in which guest users that need not register are only allowed to browse the site. A guest user can browse items by category or region, search items by region or category and view information about an item, a user, or the bidding history of an item.

RUBiS provides the capability to use three mechanisms for generating dynamic content: PHP, Java servlets, and Enterprise Java Beans (EJB). In our prototype, we use the servlet-based implementation. Each node in our prototype runs Tomcat, which delivers both static, as well as dynamic content.

Each node runs also a mySQL server ([100]) which provides access to the entire RUBiS database. This database contains some 33,000 items for sale, distributed among 40 categories and 62 regions. There is an average of 10 bids per item, or 330,000 entries in the bids table. The users table has 1 million entries. The total size of the database, including indices, is 1.4GB.

For our experiments, we have kept the default configuration, in which RUBiS does not use a database connection pool, but instead creates a new connection.
11.2. EVALUATION

Table 11.1: Profiles of the RUBiS servlets.

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>CPU Use (cycles)</th>
<th>Memory Use (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>index.html</td>
<td>250000</td>
<td>10</td>
</tr>
<tr>
<td>browse.html</td>
<td>250000</td>
<td>10</td>
</tr>
<tr>
<td>BrowseCategories</td>
<td>7500000</td>
<td>25</td>
</tr>
<tr>
<td>BrowseRegions</td>
<td>7500000</td>
<td>25</td>
</tr>
<tr>
<td>BrowseCategoriesByRegion</td>
<td>7500000</td>
<td>25</td>
</tr>
<tr>
<td>ViewItem</td>
<td>8750000</td>
<td>40</td>
</tr>
<tr>
<td>ViewUserInfo</td>
<td>8750000</td>
<td>40</td>
</tr>
<tr>
<td>ViewBidHistory</td>
<td>8750000</td>
<td>40</td>
</tr>
<tr>
<td>SearchItemsByCategory</td>
<td>15000000</td>
<td>180</td>
</tr>
<tr>
<td>SearchItemsByRegion</td>
<td>3000000</td>
<td>180</td>
</tr>
</tbody>
</table>

to the database for every new request. Activating the connection pool capability, together with a more efficient implementation of the entry points (such as [23]) would greatly increase the service rate of the system. However, such optimizations have not been carried out for this version of the prototype.

**RUBiS Servlet Profiling**

In our experiments, the nodes can deliver two static pages (index.html and browse.html) and can run seven servlets that generate dynamic content. In order to profile each such operation, we have generated (on one node) a steady flow of requests and measured the average number of CPU cycles needed by each request and the amount of memory needed by each servlet or static page. The results are given in Table 11.1.

The nodes in our testbed have 128 or 256 MB of memory. Therefore, the SearchItemsByCategory and SearchItemsByRegion servlets cannot run on some of the nodes, as they require more memory than is available.

The experiments described below use the CPU and memory requirements listed in Table 11.1.

**System Configuration**

The system is configured in such a way that each node has three overlay neighbors. The threshold for CPU overload $\text{cpu}_{\text{max}}$ is set to 85%. The number of request processing threads (in Tomcat) on a node is limited to 80, out of which 10 are for executing requests and the rest are for forwarding requests. For Overlay Construction, each node runs a protocol cycle of GoCast every 1 sec, and a protocol cycle of Cyclon every 5 sec. Each node executes the control cycle of the application placement mechanism every 60 sec, and it refreshes its routing table every 10 sec. The
size of the routing table is limited to three columns. Each request can be routed at most two times inside the cluster.

During the measurements, we retrieve from the system every 20 seconds six metrics: the arrival rate (measured in Web Interactions Per Second, or WIPS), the satisfied demand (i.e., the ratio between the service rate and the arrival rate), the average control load on a node (in messages per second), the number of configuration changes on a node (i.e., servlet starts and stops), as well as the average and the maximum response times for the incoming requests. The arrival rate, the average and the maximum response times are measured on the entry points. The other metrics are measured on the nodes.

Scenarios and Measurements

We generate different workloads with RUBiS and evaluate our testbed implementation according to three criteria. Efficiency captures the capability of the system to exhibit high performance in steady state. Adaptability captures the capability of the system to respond to a change in the operating conditions by rapidly reconfiguring and converging to a new steady state. Manageability refers to the capability of the system to dynamically adjust its configuration to changes in management policies.

System Performance as a Function of the External Load

In this series of experiments, we measure the service rate of the system, i.e., the maximum number of requests that the system can handle, while gradually increasing the external load. Each point in the graphs shown in Fig. 11.3 contains the result of an entire run on the testbed. In each run, the load remains constant.

As can be seen in Figs. 11.3(a) and 11.3(b), the service rate (in req/sec) increases linearly with the number of emulated clients, up to around 4900 emulated clients, at which point the system starts dropping requests. Above 4900 emulated clients, the arrival rate approaches the system capacity and the service rate of the system becomes flat, which is a desirable behavior of a system under overload. Fig 11.3(c) shows that the average and the maximum response times increase until the number of emulated clients reaches 4500. Beyond this point, the average and the maximum response times tend to decrease. We explain this decrease by the fact that the system drops a larger number of requests, coupled with the way in which requests are generated in RUBiS. After an Emulated Client has unsuccessfully invoked a specific servlet three times in a row, it restarts at index.html. Serving this static page requires less CPU processing power than generating a page with dynamic content.

Finally, as seen in Fig. 11.3(d), the control load per node remains constant, while the number of configuration changes per node stays very small.
11.2 EVALUATION

Figure 11.3: System performance as a function of the external load, measured as the number of emulated clients.

**System Adaptability**

In this experiment, we test the capability of the system to adapt and reconfigure in response to the addition of new applications, combined with an additional external load for those applications. Initially, the nodes deliver two static pages (index.html and browse.html) and dynamic content through three servlets (BrowseCategories, BrowseRegions and BrowseCategoriesInRegion). The initial request arrival rate is 200 req/sec.

After two minutes, four new servlets are started (SearchItemsByCategory, ViewItem, ViewBidHistory and ViewUserInfo) and an additional load of 300 req/sec is created for those new servlets. As Fig. 11.4(a) shows, the system needs about two minutes (i.e., about two control cycles of the application placement mechanism) to reconfigure, before the satisfied demand converges to 100%. We see in Fig. 11.4(b) that the
CHAPTER 11. IMPLEMENTATION OF A SERVICE MIDDLEWARE THAT SCALES IN SYSTEM SIZE AND APPLICATIONS

![Graphs showing system adaptability](image)

Figure 11.4: System adaptability to the addition of new servlets and an increase in load for these servlets. The x-axis indicates wall-clock time.

satisfied demand for the already running servlets temporarily decreases during the transient period, while the satisfied demand for the new servlets increases almost linearly in time. Fig. 11.4(c) shows that the average and the maximum response times increase immediately after the change in the external load, but eventually stabilize around slightly higher values. This increase in the response times is expected and consistent with Fig. 11.3(c), as higher arrival rates correspond to longer response times. Finally, we see from Fig. 11.4(d) that the system stabilizes after about 4 minutes, and that the nodes do not make any changes in their configuration later in time. The control load on a node increases, because the number of active servlets increases as well.
11.2 EVALUATION

![Graphs](image)

Figure 11.5: System Manageability: Providing Differentiated Service.

**System Manageability: Providing Service Differentiation**

This experiment tests the capability of the system to provide differentiated service under overload, in response to changes to the relative importance of each application performed by an administrator. In this scenario, the nodes deliver static content (for the index.html and browse.html pages) and dynamic content for seven servlets (BrowseCategories, BrowseRegions, BrowseCategoriesInRegion, ViewItem, ViewBidHistory, ViewUserInfo and SearchItemsByCategory). The arrival rate is around 800 req/sec and the system is in overload (see Fig. 11.5(a)).

After two minutes, the importance factor for all servlets, except the SearchItemsByCategory servlet, is steadily increased by 1 every 20 seconds. As a consequence of this management operation, we observe that two out of the four nodes that were originally providing the SearchItemsByCategory servlet stop offering it and start offering other servlets instead. Fig. 11.5(b) shows that this results in a decrease in
CHAPTER 11. IMPLEMENTATION OF A SERVICE MIDDLEWARE THAT
SCALES IN SYSTEM SIZE AND APPLICATIONS

the satisfied demand for SearchItemsByCategory and an increase in the satisfied demand for the other servlets.

Fig. 11.5(a) shows that the overall utilization of the system decreases, as the servlets with a higher importance require less CPU resources than SearchItemsByCategory. Moreover, a RUBIS client emulator will request one of the CPU-intensive servlet (ViewItem, ViewBidHistory and ViewUserInfo) after it has issued a request for SearchItemsByCategory. As a large number of requests for the SearchItemsByCategory servlet fail, the clients generate more requests for the static content and the BrowseCategories/BrowseRegions servlets. (After an Emulated Client has requested a specific servlet three times in a row without succeeding, it restarts its session by refreshing index.html.)

Fig. 11.5(c) shows that there are no significant changes in the response times. Actually, the average response time decreases slightly after the changes in the importance factor of each servlet. We explain this decrease by the fact that lowering the importance factors assigned to the CPU-intensive requests decreases the number of such requests that are processed by the system. Finally, Fig. 11.5(d) shows that the control load per node remains almost constant during the entire experiment.

11.3 Discussion

In this report, we have presented the testbed implementation of the design described in Chapter 10. We have evaluated the prototype by running the RUBIS benchmark. We have shown that the system dynamically and effectively reconfigures, in response to load changes, in response to the addition of new applications and in response to changes in the operating policies.

An interesting observation from the first scenario is that the service rate of the system increases linearly with the offered load, up to the limit of its service capacity. In overload, the service rate of the system remains flat. Moreover, the average CPU utilization in the system is close to 85%, which is the threshold above which a node becomes overloaded. Overall, these results show that the prototype behaves similarly to the simulated design, therefore validating our design and proving its feasibility.

Acknowledgment

The measurements in this section were performed by Erik Häggbrink, as part of his Master Thesis project at KTH.
Bibliography


Curriculum Vitae

Constantin Adam is a Ph.D. candidate at the Royal Institute of Technology, Stockholm, Sweden. His research interests include self-organizing systems, and scalable service management.

He received his B.Sc. in computer science from the Swiss Federal Institute of Technology, Lausanne, Switzerland in 1996, and his M.S. in electrical engineering from Columbia University, New York, USA, in 1998.

He has worked as a Research Engineer at Columbia University from 1996 until 1998 and as a Senior Software Engineer from 1998 until 2002 for Xhind, Inc. a high technology startup located in Manhattan.

He has had two internships with IBM. During the Summer of 2005, he developed a distributed algorithm for application placement and filed a Patent Application: C. Adam, G. Pacifici, M. Sreitzer, M. Steinder, C. Tang, "Decentralized Application Placement for Web Application Middleware". During the Summer of 2006, he has developed an algorithm for electing and managing process groups that provide high-availability services in a large-scale system.