BUILDING AND MANAGING LARGE SCALE DISTRIBUTED SERVICES

BY

JIN LIANG

B.E., University of Science and Technology of China, 1998
M.E., Institute of Software, Chinese Academy of Sciences, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois
Abstract

Recent research in peer-to-peer and grid computing has made it possible to build Internet scale services such as content distribution, storage service, name service and publish/subscribe. By utilizing large number of service nodes that collaborate in a decentralized fashion, such services can potentially achieve high scalability, availability, reliability and QoS/performance. Despite such potential, building large distributed services and testing them in a real world, widely distributed environment remains a difficult task. This is because first, a wide area environment is full of various network and node failures. Therefore, services targeting such environment must have built-in mechanisms to deal with such failures. Further, such mechanisms must not rely on centralized control, due to the scale of the services. Second, running services in a wide area environment requires system support for deploying, monitoring and controlling the services. However, current computing infrastructures generally lack powerful tools for managing widely distributed services. As a result, service developers often have to resort to ad hoc methods for service management.

In this dissertation we present our research aimed at simplifying the development of large scale distributed services. Large scale services are a special class of large distributed applications. As a result, we focus on addressing the challenges involved in the design, implementation and management of such applications. We first present OCMA, a layered architecture for designing large distributed applications. OCMA divides such applications into three layers: the membership layer that keeps up-to-date information about other nodes in the system; the overlay layer that builds or maintains the overlay structure; and the application layer that carries out the application specific processing. Such functional decomposition not
only simplifies the design of service applications, but also facilitates the reuse of components (layers) and the innovation within each component. For example, we have designed two large distributed applications, the DagStream system for locality aware P2P streaming and the Management Overlay Networks (MON) system for distributed management. Both are designed according to the OCMA architecture, and both have explored novel techniques for some of their layers.

Through the implementation of multiple large scale applications, we have extracted a C++ framework called PPF (Protocol Plugin Framework) that can be reused to implement large distributed applications. Using PPF, application developers only need to implement the high level protocol between different application nodes. When the protocol is plugged into PPF, the same code can run in both simulation and real world mode. This minimizes the possibility of introducing bugs when porting simulation code to real world deployment.

MON is not just an example application designed according to OCMA, but also a simple, scalable and lightweight tool we have built for managing service applications running in a wide area environment. MON facilitates the management of such applications by building short-term, on-demand overlay networks that can be used to instantly query and control the distributed application status. Such distributed status query and control allows application developers to quickly detect, diagnose and correct potential application problems. In addition to MON, we have also explored algorithms that can adaptively combine continuous monitoring and dynamic query in order to minimize information management overhead.

The major contributions of this dissertation are as follows. First, we present a layered architecture called OCMA and several design techniques such as on-demand overlay construction and control plane services for designing large scale distributed applications. Second, we provide PPF, a C++ framework that can be reused to implement such applications. Third, we build MON, a powerful tool for dynamically querying and controlling the status of distributed service applications. Such dynamic query and control can facilitate the detection and diagnose of potential application problems.
To my family.
Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Klara Nahrstedt, for her invaluable guidance and continuous support throughout my PhD research. Her wide knowledge and key insights have been a constant help during my pursuit of research, and her patient advice and thoughtful directions have greatly improved my research, writing and presentation skills, which will surely benefit my future career. I especially appreciate her generosity in time and willingness to talk whenever I asked, usually out of her busy schedules. I feel extremely grateful to have an advisor who not only advises and supports, but also cares about my achievements and my success.

I would also like to thank Professor Indranil Gupta. I have been working closely with him on the MON project. Many of his ideas and comments have helped me develop the project, and his advice on research and academic writing has been extremely helpful. My sincere gratitude also goes to my other honorable committee members, Professor Roy Campbell and Professor Yuanyuan Zhou. Their discussions and feedbacks have helped improve my research work. In addition, Professor Campbell has offered me a lot of help on various joint projects between the SRG group and the MONET group, and Professor Zhou has given me valuable advice on publication and career development.

Many thanks go to my colleagues in and out of the MONET research group. In particular, I would like to thank Wanghong Yuan, Xiaohui Gu, Jingwen Jin, Steven Ko and Long Vu for their fruitful discussions and collaborations on research projects. I am grateful to Li Xiao, Kai Chen, Vanish Talwar, Wenbo He, Ying Huang, William Conner, Thadpong Pongthawornkamol and Jay Patel and for their discussions and insightful feedbacks on my
research, and I thank Yi Cui, Bin Yu, Yuan Xue, Muyuan Wang, Hoang Nguyen, Wanmin Wu, Zhengyu Yang, Won J. Jeon and Samarth Shah for their help on various MONET projects and events. I have been truly fortunate to work with them and I cherish their warmth and friendship.

I am also grateful to Anda Ohlsson, Barb Cicone, Mary Beth Kelley, Erna Amerman, Shirley Finke and other department staff for their administrative and technical support. Their work has helped smoothe my PhD process.

Last but not least, I would like to thank my entire family for their love and moral support, without which all my achievements would not have been possible. I would especially like to thank my parents for their dedication and the values they have passed on to me, my wife Yiyi Zeng for her love and understanding, and my son Andrew for the happiness he has brought to the whole family. This thesis is dedicated to them.
# Table of Contents

List of Tables ................................................................. xi
List of Figures ................................................................. xii

Chapter 1  Introduction .................................................. 1  
   1.1 Background and State of the Art ................................. 2  
   1.2 Simplifying Development of Large Distributed Service Applications .......................... 4  
      1.2.1 Simplifying Application Design .......................... 5  
      1.2.2 Simplifying Application Implementation .............. 5  
      1.2.3 Simplifying Application Management .................. 6  
   1.3 Thesis Contribution .............................................. 6  
   1.4 Thesis Outline ................................................... 7  

Chapter 2  System Model .................................................. 9  
   2.1 Application Model .............................................. 9  
   2.2 Network and Communication Model ............................. 10  
   2.3 Failure Model .................................................. 12  
   2.4 Management Tasks Model ....................................... 13  

Chapter 3  Design Architecture and Techniques for Service Applications 16  
   3.1 OCMA: A Layered Architecture for Large Distributed Applications ......................... 16  
      3.1.1 Membership Management ................................. 17  
      3.1.2 Overlay Construction & Maintenance .................. 19  
      3.1.3 Application Specific Processing ...................... 19  
      3.1.4 Discussions ............................................ 20  
   3.2 Novel Techniques for Designing Large Distributed Applications ......................... 21  
      3.2.1 On-demand Overlay Construction ...................... 21  
      3.2.2 Control Plane Services ................................ 23  
   3.3 DagStream: Locality Aware and Failure Resilient P2P Streaming ......................... 24  
      3.3.1 Existing P2P Streaming Systems ....................... 24  
      3.3.2 Streaming with DAG Overlays ......................... 26  
      3.3.3 Experimental Results .................................. 32
Chapter 4  A Reusable Framework for Implementing Service Applications  42
  4.1 PPF Overview .................................................. 42
  4.2 Time and Network Simulation ................................. 45
  4.3 Support for Application Evaluation and Management ........ 47

Chapter 5  Management Overlay Networks for Service Applications . . . 49
  5.1 MON Overview .................................................. 49
  5.2 Gossip Based Membership Management ..................... 51
    5.2.1 Membership Gossip Protocol ............................. 52
    5.2.2 Analysis of the Membership Protocol .................. 53
  5.3 On-demand Overlay Construction ............................ 59
    5.3.1 General Algorithm ....................................... 59
    5.3.2 The \texttt{randk} Algorithm ............................ 61
    5.3.3 The \texttt{leafset+RF} Algorithm ....................... 61
    5.3.4 The \texttt{tree+RF} Algorithm .......................... 64
    5.3.5 On-demand DAG construction ........................... 65
  5.4 Application Management with MON ........................... 66
    5.4.1 MON Query Language ..................................... 66
    5.4.2 MON API and Scripts ................................... 68
    5.4.3 Querying the Application Internal Status .............. 69
    5.4.4 Distributed Log Query ................................. 70
  5.5 Evaluation ................................................... 70

Chapter 6  InfoEye: Self-Conﬁguring Information Management ............ 75
  6.1 InfoEye Overview .............................................. 75
  6.2 The InfoEye Model ............................................ 77
    6.2.1 InfoEye Queries ......................................... 77
    6.2.2 Statistical Patterns .................................... 78
    6.2.3 Problem Formulations .................................. 80
  6.3 Design and Algorithms ...................................... 82
    6.3.1 Push Attribute Selection ................................ 83
    6.3.2 Push Threshold Conﬁguration ........................... 84
    6.3.3 Push Interval Selection .................................. 85
    6.3.4 Practical Issues ......................................... 86
  6.4 Experimental Evaluation .................................... 86
    6.4.1 Evaluation Methodology .................................. 86
    6.4.2 Simulation Results ....................................... 88
    6.4.3 Prototype Results ....................................... 91
  6.5 Discussions .................................................. 95

Chapter 7  Related Work ............................................ 96
  7.1 Structured and Unstructured Overlay Networks .................. 96
  7.2 Gossip Protocols ............................................. 97
  7.3 Implementing Large Distributed Applications ................ 98
  7.4 P2P Media Streaming ......................................... 99
7.5 Information Monitoring and Distributed Application Management 100

Chapter 8 Concluding Remarks 102
  8.1 Conclusion 102
  8.2 Future Work 103

References 105

Author’s Biography 111
List of Tables

2.1  Example network and node failures on the PlanetLab  . . . . . . . . . .  12
6.1  Notations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .  77
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Service application development phases and our contributions</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Overlay network</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>The OCMA layered architecture</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Overlay network with poor connectivity (failure resilience)</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>DagStream streaming overlay</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Layered design of DagStream peers</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>The RandPeer membership service</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Mapping from membership bin to RandPeer nodes</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Multi-parent, receiver-driven streaming</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Parent-child delay of different parent discovery techniques</td>
<td>34</td>
</tr>
<tr>
<td>3.9</td>
<td>Level comparison with MST</td>
<td>35</td>
</tr>
<tr>
<td>3.10</td>
<td>Effect of different parent selection policies</td>
<td>36</td>
</tr>
<tr>
<td>3.11</td>
<td>Effect of delay threshold $d$</td>
<td>37</td>
</tr>
<tr>
<td>3.12</td>
<td>Streaming quality delivered by DagStream</td>
<td>38</td>
</tr>
<tr>
<td>3.13</td>
<td>Performance results on about 228 PlanetLab nodes</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Components of the PPF framework</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Components of an example application with three layers</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Main dispatcher loop for real time, real network</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>Main dispatcher loop for simulated time and network</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Management overlay network</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>MON daemon and application node</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>$birth_time$ maintenance</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>Membership view maintenance protocol</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>Average age of membership entries</td>
<td>57</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison of in-degree distribution</td>
<td>58</td>
</tr>
<tr>
<td>5.7</td>
<td>General tree construction algorithm</td>
<td>60</td>
</tr>
<tr>
<td>5.8</td>
<td>Leafset</td>
<td>62</td>
</tr>
<tr>
<td>5.9</td>
<td>Segments in membership tree</td>
<td>65</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison of the tree construction algorithms</td>
<td>71</td>
</tr>
<tr>
<td>5.11</td>
<td>Comparison results on the PlanetLab</td>
<td>73</td>
</tr>
<tr>
<td>5.12</td>
<td>Session reliability of tree and DAG overlays</td>
<td>73</td>
</tr>
</tbody>
</table>
Many large scale distributed applications have emerged recently that provide various Internet scale services. Examples include content distribution [3], storage service [26, 71], name service [61, 64] and publish/subscribe [63, 19]. These services often involve large number of application nodes that collaborate with each other in a decentralized fashion. As a result, they can potentially improve the service quality by re-directing clients to nearby servers, routing data around network faults, and sharing load among the service nodes. Despite such potential benefits, building large scale service applications requires novel design techniques for achieving scalability, failure resilience and high QoS/performance, and running the applications in a wide area environment involves even more challenges associated with deploying, monitoring and controlling the applications.

The above challenges have motivated our research into the design techniques and architectures, reusable code bases, and management tools that are aimed at simplifying the design, implementation and management of large distributed applications for Internet scale service provisioning. As is shown in Figure 1.1, the development of service applications involves three phases: design, implementation and real world deployment/management. Our research has contributed to all three phases. In the design phase, we have explored different design techniques and proposed a layered architecture for designing large distributed service applications. In the implementation phase, we have created a reusable C++ framework for application implementation. And in the deployment and management phase, we have built management tools that facilitate the monitoring and controlling of service applications running in a wide area environment. Although our work focuses on large distributed appli-
In the last several years, many “peer-to-peer” (P2P) applications have emerged on the Internet and enjoyed great popularity. Examples are file sharing applications such as Napster [8], Gnutella [13] and KaZaA [7], content distribution such as BitTorrent [2], and media streaming such as CoolStreaming [79], PPLive [10] and PPStream [11]. For all these applications, users can join and leave the system at any time. When they are in the system, each peer not only consumes service, but also provides service to other peers. This is a significant departure from traditional client/server applications, where the server is the only service provider and the clients are the only service consumers.

Although some P2P applications have raised concerns about Internet users sharing illegal contents, the underlying technology (P2P technology) has generated great interest in the research community, because it can potentially be used to achieve high scalability, availability, reliability and QoS/performance for large distributed applications that pro-
vide Internet scale services. In fact, significant research efforts have gone into the design of many such service applications using P2P technology, including distributed hashtables (DHTs) [74, 70, 65, 80, 35, 53, 55, 37], which provide efficient keyword lookup in a large distributed system; application level multicast [39, 47, 15, 75], which distribute media data to large number of receivers; and others such as name service [64], indirection service [73], resource discovery [58] and publish/subscribe [19].

Despite the potential benefits and numerous research results, building large distributed service applications and running them in a real world, widely distributed environment remains a difficult task, due to the lack of design principles, software architectures, re-usable code bases and management tools for such applications.

In terms of application design, existing research has proposed distributed hashtables [74, 70, 65, 80] (DHTs) as a “common substrate” upon which other applications can be built. DHTs provide decentralized message routing and keyword lookup. Such powerful mechanisms may allow applications to be built with great ease. In reality, however, this has not been very successful for several reasons. First, DHTs impose performance overhead because each message is routed via multiple hops. Second, DHTs need to maintain certain invariants in order to work properly. For example, if two nodes have adjacent node IDs in a DHT, they must have consistent views on their routing tables [21]. Consistent routing is critical to the correctness of DHTs, and its implementation may impose performance penalty on DHT message routing. However, for many applications, such strong guarantee may not be necessary.

In terms of application implementation, some research projects have provided systems such as MACEDON [69] and P2 [52], which allow application developers to implement their systems using a high level, domain specific language. Such high level implementation is then automatically translated into C++ code that can run in a real world environment. The drawback, however, is that the high level language may present a high learning curve, and the automatic translation may make it difficult for application developers to fine tune their
code for performance optimization. Some systems such as FreePastry [6] provide support for both simulation and real world execution. However, the APIs that they provide are DHT APIs, which are different from the socket APIs that most system developers are familiar with.

In terms of application management, existing tools [25, 5] have focused on managing the distributed computing infrastructure rather than the applications running on top of it. For example, they often continuously monitor a predefined set of system level metrics such as CPU load and free memory, but have not provided the ability to monitor the internal state of an application process or the log files generated by the process. In addition, these systems are mostly based on a centralized architecture, thus may have scalability problems when the application size increases. Some systems [9, 14] have attempted to support application management in a widely distributed environment. However, they have focused on coarse grained monitoring and do not provide the ability to dynamically query and control the application status.

1.2 Simplifying Development of Large Distributed Service Applications

The difficulty to build and manage large distributed service applications has motivated our research into simplifying the design, implementation and management of such applications. Specifically, we have investigated various design techniques, software architectures, reusable code bases and management tools for developing large distributed applications. Note although we focus on large distributed service applications, some of our results (e.g., design architecture and implementation framework) also apply to other applications such as P2P applications.
1.2.1 Simplifying Application Design

To simplify the design of large distributed applications, we have taken an empirical approach. We have designed two concrete applications, one for distributed management and the other for P2P media streaming, and explored novel design techniques for these applications. Our techniques such as on-demand overlay construction and control plane services can potentially benefit the design of other large distributed applications. In addition, from our design experience, we have discovered a layered Overlay Construction and Maintenance Architecture (OCMA) for large distributed applications. OCMA divides such applications into three layers: the membership layer that keeps up-to-date information about other nodes in the system; the overlay layer that builds or maintains the desired overlay structure; and the application specific layer that carries out application specific processing. Such a layered architecture not only simplifies application design, but also facilitates the reuse of different software layers. For example, two applications may share the same membership management scheme even though they may have different overlay structures and application specific processing.

1.2.2 Simplifying Application Implementation

Through our implementation of several large distributed applications, we have extracted a C++ framework called Protocol Plugin Framework (PPF) that can be reused to implement service applications in a single thread, event driven style. The framework provides event scheduling and asynchronous network I/O for abstract protocol modules. Thus a service developer can focus on his or her concrete protocol modules. Once the protocol modules are plugged into the framework, the whole system can automatically execute in both simulation and real world mode. This greatly facilitates the initial debugging of distributed protocols and the transitioning from simulation mode to real world deployment. The event driven style also means multiple protocol modules can be easily composed together, thus it is ideal
for implementing service applications that are structured into multiple layers.

1.2.3 Simplifying Application Management

When a service application is running in a wide area environment, it is important for the service developer to closely monitor the application execution in order to detect potential problems, and to take corrective actions when some failures are detected. This can be difficult if there are hundreds or thousands of application nodes to monitor, and if the computing infrastructure (e.g., PlanetLab [62]) does not have sophisticated support for application level management.

We have developed the Management Overlay Networks (MON) [49] system to ease service application management. MON facilitates application management in several aspects. First, it provides the ability to dynamically query and control the distributed application status, such as the internal state of the application process and the log file generated by the process. Second, MON provides built-in mechanisms for information aggregation. As a result, the service developer can get aggregate information such as top K, average and histogram instead of the raw data. Third, we have implemented a SQL-like query language and made it available through some C++ programming API. As a result, service developers can easily integrate MON queries with higher level programming logics.

In addition to MON, we have also designed a self-configuring information monitoring system called InfoEye [48] that can adaptively combine dynamic query and continuous monitoring to minimize information management overhead.

1.3 Thesis Contribution

This thesis makes the following major contributions:

- We present OCMA, a layered architecture for designing large distributed service applications. OCMA divides such applications into smaller components (layers). Thus it not
only simplifies application design, but also facilitates component reuse and innovation within each layer. We have also designed novel techniques such as on-demand overlay construction and control plane services for building large distributed service applications. Although our focus is on service applications, our architecture and techniques can also be used for other large distributed applications such as P2P applications.

• We have created PPF, a reusable framework for implementing large distributed service applications. PPF simplifies application implementation because it frees the application developer from the details of asynchronous network programming, and it allows the same code to run in both simulation and real world mode. The latter is especially useful for the initial debugging of the service application, and for the transitioning from simulation to real world deployment.

• We have built the Management Overlay Networks (MON) system, a tool for managing services running in a widely distributed environment. MON provides the ability to dynamically query and control the status of a distributed service application. As a result, service developers can quickly detect potential application problems and take control actions. MON is currently deployed on the PlanetLab [62] and offers public service to the PlanetLab community. Since its deployment, hundreds of users have used MON for PlanetLab status query. We have also expanded the dynamic query capability of MON and built the InfoEye system that can adaptively combine dynamic query and continuous monitoring in order to minimize information management overhead.

1.4 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2 we describe our system model. Chapter 3 presents the layered overlay construction and maintenance architecture (OCMA) for designing large distributed applications. It also discusses two novel design techniques
and presents the design of one specific application, the DagStream system for locality aware P2P media streaming. Chapter 4 describes the PPF framework that can be reused for application implementation. Chapters 5 and 6 present two management tools we have built for managing distributed services, namely the MON system and the InfoEye system. Finally, Chapter 7 distinguishes our work from related research, and Chapter 8 concludes the paper with discussion on future work.
Chapter 2

System Model

The goal of our research is to simplify the process of designing, implementing, deploying, monitoring and controlling large distributed applications that provide Internet scale services. This chapter introduces our application model, network and communication model, failure model and management task model.

2.1 Application Model

There are two classes of large distributed applications, *infrastructure based* and *end host based*. Infrastructure based applications are often provided and managed by a single organization. The applications are deployed in a distributed computing infrastructure such as the PlanetLab [62], and their goal is to provide networked service to their clients (Internet users or other applications). These applications are the focus of this thesis and we call such applications *service applications*. We will omit the word “service” when the meaning is clear.

Service applications typically consist of a large number $N$ of application nodes (hundreds up to hundreds of thousands, or even more). The application nodes are distributed, possibly across wide area networks. Different application nodes are symmetric in terms of functionality, and they communicate with each other in a decentralized fashion. Examples of service applications include content distribution [1, 3], name service [61, 64], storage service [26, 71], indirection [73], and resource discovery service [58].

End host based applications are also called peer-to-peer (P2P) applications. Different from service applications, P2P applications are not provided and managed by a single entity.
Instead, the application nodes are autonomous end users that may join and leave the system at any time. Such frequent join and leave can cause high instability for P2P applications. For example, Chu et. al [38] have shown that for some media broadcasting events, the median session life time of a node can be as short as 7 minutes, and mean session life time as short as 11 minutes. Although our focus in this dissertation is on infrastructure based service applications, some of our results, especially the design architecture and implementation framework can also benefit the development of P2P applications \(^1\).

We assume the nodes of service applications are widely distributed. Some nodes may be clustered in local area networks. However, each cluster only contains small number of nodes. In other words, we assume the target environment of service applications is more like the Akamai [1] content distribution network or the PlanetLab [62] testbed, rather than a small number of large data centers. Assuming a wide area environment makes our model more general. It is likely that service applications and management tools based on our assumptions will still work in an environment with small number of data centers, although the data center environment may offer additional opportunities for performance improvement.

2.2 Network and Communication Model

We assume nodes of a service application can communicate with each other by sending/receiving messages. A node \(p_i\) can send message to any node \(p_j\), as long as it has the contact information of \(p_j\) (e.g., its IP address and port number). Due to scalability, however, each node \(p\) can only maintain contact information for \(m\) other nodes, with \(m \ll N\). These \(m\) nodes are called the membership view of \(p\).

Although a node \(p\) can send messages to any node in its membership view, at any given time, \(p\) may be actively communicating with only a subset of the nodes in its membership

\(^1\)Note we should differentiate P2P applications from P2P technology. P2P applications are large distributed applications that consist of autonomous end users (in contrast to service applications), while P2P technology refers to the decentralized communication mechanisms underlying large distributed applications (both service and P2P applications).
These nodes are called its overlay neighbors, and such neighboring relationship forms an overlay network that may be independent of the underlying physical network. Figure 2.1 shows an example application. The bottom part is the physical network, and the top part is the overlay network.

The topology of the overlay network is important to the application performance. For example, a tree topology is often scalable and efficient, because the height of the tree is logarithmic to the system size, and there are no redundant connections between nodes. However, from a failure resilience point of view, a tree structure may be vulnerable to failures, because a single node or network failure could lead to system partitioning.

We assume the application nodes are distributed across wide area networks. Therefore, each overlay link between two nodes corresponds to a path in the underlying IP network. This means although nodes can directly connect to each other, the connections may have different characteristics such as delay and network bandwidth. Ideally, a node should preferably connect to nearby nodes, so that efficient network resource usage can be achieved.

The overlay structure of an application is not static. It evolves over time as existing nodes fail (crash or leave the system) or recover. When a node fails, its neighbors will be able to
detect the failure. In this case, either the neighbors locate alternative neighbors, in order to maintain the overlay topology, or a completely new overlay is constructed, depending on the particular applications.

2.3 Failure Model

<table>
<thead>
<tr>
<th>date</th>
<th>nodes</th>
<th>symptom</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/2005</td>
<td>planetlab2.nbgisp.com</td>
<td>disk error caused the CoMon service to be inaccessible</td>
</tr>
<tr>
<td>08/2005</td>
<td>planetlab2.ucb-dsl.nodes.planet-lab.org</td>
<td>system out of memory caused “sendto(): No buffer space available”</td>
</tr>
<tr>
<td>09/2005</td>
<td>planetlab1.uc.edu and seu1.6planetlab.edu.cn</td>
<td>routing loop caused connectivity problem between the two nodes</td>
</tr>
<tr>
<td>03/2006</td>
<td>planetlab1.cs.uiuc.edu and dlut2.6planetlab.edu.cn</td>
<td>persistent high loss rate (76%) and large rtt (about 300ms)</td>
</tr>
<tr>
<td>03/2006</td>
<td>planetlab1.cs.uiuc.edu and planet2.njit.edu</td>
<td>very large persistent rtt (about 4 seconds) and large loss rate (14%)</td>
</tr>
<tr>
<td>03/2006</td>
<td>pli2-pa-3.hpl.hp.com and planetslug3.cse.ucsc.edu</td>
<td>high loss rate (70%) but very small rtt (about 4ms)</td>
</tr>
</tbody>
</table>

A significant challenge in both the design and management of large distributed applications is to deal with various network and node failures in a widely distributed environment. Network failures include packet delay, packet loss, packet reordering and network path outage. Although packet delay and loss may also occur in local area networks, in a wide area environment, the packet delay and loss can be much higher. For example, on the PlanetLab [62] testbed, we have observed node pairs between which the packet loss rate could be up to 70% and the message delay could be up to several seconds (as shown in Table 2.1). Such high loss rate and delay means the design of an application must explicitly take such failures into account (e.g., use larger timeout values, or better yet, find alternative routes between two nodes). Another type of network failures are routing problems [31, 57], which may cause the network path between two nodes to be unusable, even though both nodes are alive. One consequence of this is that even if a node cannot directly communicate with some
other node, it cannot conclude that the node has failed; neither can it notify other nodes about such “node crashes”.

Applications running in a distributed environment may also encounter various node failures. For example, an application may want to load some shared library. However, the library may not be present on some of the computer nodes. This could happen, if there are many computer nodes and some of them have missed the recent software update. As another example, an application may want to write some data to the disk. However, the disk space on some nodes may have been used up, due to bugs of some other applications running on the same node. As a result, the “write” will fail. As yet another example, when an application needs to bind to some network port on hundreds of computers, it is possible that the binding will succeed on all but a few nodes, where the network port has been taken by some other applications. Some of the node failures that we have seen on the PlanetLab are shown in Table 2.1. For many of these environment caused node failures, the application may not be able to recover on its own. As a result, it is important to have powerful management tools that can help the application developers quickly detect and diagnose such problems.

Since some application nodes may be clustered together, there may also be correlated failures. For example, the communication between two clusters of nodes may go through the same network path, and the failure of this path may cause one cluster to appear as correlated failures to the other cluster. We assume correlated failures can happen. However, we address such failures implicitly For example, our on-demand overlay construction technique addresses such failures by making them irrelevant, because it is likely no overlay is being used/maintained when the failures happen.

\section{Management Tasks Model}

The management of a large distributed service application involves many management tasks. First, the application developer needs to deploy the application on the target environment.
This often means copying the application code to hundreds or even thousands of distributed computers; second, the application developer needs to closely monitor the application status and control the application execution if needed. We have explored application deployment in our early work on MON [49]. However, application deployment is similar to content distribution, for which there has already been much research [3, 4]. Therefore, in this dissertation, we focus on the monitoring and controlling of distributed application status.

When a service application is running in a wide area environment, it is important to closely monitor and control its status in order to detect and correct any potential application problems. The status of an application is the aggregate status of all its application nodes. Each application node has two kinds of status: external and internal. External status of an application node can be observed without support from the application process. For example, the CPU, memory and disk usage of the underlying computer, the resource usage of a particular process or file, and the contents of the log file generated by the process. Internal status can only be obtained through the application process. For example, the set of overlay neighbors that an application node currently has, the number of service requests that the node has processed, and the percentage of media blocks that are received before their playback deadline.

Since each application node may have hundreds of status metrics, it is impossible to continuously monitor all these metrics. Therefore, the ability to dynamically query the application status is needed. This means when an application developer needs to obtain certain status metric, he or she can issue a request for the metric. This request is dynamically executed and the desired information is returned. Compared with continuous monitoring, dynamic query is more flexible because it is not limited to a predefined set of status metrics.

Even for a single status metric, due to the large number of application nodes, it may be inconvenient to present detailed data from each individual node to the application developer. Thus, the ability to aggregate the status metric is highly desirable. For example, returning the top 10 application nodes using the most memory, instead of the memory usage of each
individual node.

The semantics of the status metrics is highly dependent on the application being managed. Therefore, it is undesirable for the management tool to understand the metrics being queried. From the management point of view, each status metric is either a numeric value or text string. Once the metric value is obtained using some local interface on each node, the management tool should be able to aggregate the result and return it to the application developer.
Chapter 3

Design Architecture and Techniques for Service Applications

In this chapter, we present OCMA, a layered architecture for designing large distributed applications. We also discuss two possible design techniques within this architecture. The OCMA architecture is applicable to both service applications and P2P applications. For example, we have designed two large distributed applications, the DagStream system for locality aware P2P media streaming; and the Management Overlay Networks (MON) system for distributed management. Although the two applications are very different from each other, both conform to the OCMA layered architecture. We will present the design of DagStream in this chapter. The design of MON will be presented in Chapter 5.

3.1 OCMA: A Layered Architecture for Large Distributed Applications

Although recent years have seen a lot of research results in the area of large distributed applications, not much work has investigated the overall architecture for such applications. As a result, whenever a new application needs to be designed, it has to be designed from scratch due to the lack of guiding principles as to how the application can be structured and how different subsystems can be composed to achieve the application goal.

The architecture of a software system refers to a set of design principles that specify how the system can be decomposed into smaller components and how the components interact with each other. For a complex system, the importance of a good architecture is hard to

\[1\] Thus we will talk about large distributed applications in general in this chapter.
over-emphasize. For example, the tremendous success of the Internet is to a large extent due to its layered architecture. Each layer provides a different level of abstraction, with the interface between the layers clearly defined. This not only simplifies network system design, but also facilitates innovation within each layer, since the internal design of a layer is transparent to other layers as long as the interface between the layers is preserved. As another example, recent work [24, 30] has recognized the importance of an overall architecture for sensor networks and has proposed a sensor network architecture that centers on a unifying link layer abstraction.

In our research, we have designed several large distributed applications, including a locality aware P2P media streaming application called DagStream, and a dynamic distributed status query and control system called Management Overlay Networks (MON). From our design experience, we found that such applications often can be divided into three coarse grained layers as shown in Figure 3.1. The bottom layer is for membership management. The middle layer is for overlay construction and maintenance. And the top layer is for application specific processing. Both the DagStream and MON systems are designed according to this layered architecture. Below we briefly describe the functionality of each layer at a high level.

### 3.1.1 Membership Management

In a client/server application, every client only needs to know about the server. In a large distributed application, a node must know about some other nodes in the system, because it can potentially communicate with these nodes. Due to the scale of the application, it is
difficult to maintain up-to-date information about all nodes in the system. As a result, a node \( p \) may choose to maintain information about a subset of the nodes. This subset of nodes is called the *membership view* of node \( p \), and these are the nodes that it can directly communicate with. Note a node often only needs to communicate with its neighbors in the overlay layer. This is called its neighbor set and it is usually a subset of the membership view.

In a large distributed system, nodes may join and leave the system at any time. As a result, when an overlay neighbor fails or experiences degraded performance, an alternative neighbor must be quickly located. Keeping information about potential neighbors (i.e., nodes in the membership view) thus facilitates the quick recovery from neighbor failures.

Some early large distributed applications such as ESM [39] maintain full membership information about all nodes in the system. As the system increases in size, maintaining full membership would inevitably cause high overhead. Some other systems such as NICE [15] and Zigzag [75] maintain *implicit* membership information. This means the set of nodes known by a node is exactly its neighbor set. There is no explicit mechanism for membership management. This may cause slow failure recovery since when an overlay neighbor fails, a new neighbor must be dynamically discovered. Many recent systems [38, 79, 10] have used explicit protocols for membership management. Each node maintains a relatively large (but still partial) membership view, and exchanges information with each other to keep the membership up-to-date. As a result, whenever the overlay layer needs a new neighbor, it can be quickly located from the membership view.

The goal of the membership layer is thus to maintain high quality membership information. This means newly joined nodes should be quickly propagated to other nodes, and failed nodes should be quickly detected and removed. If a failed node is presented to the overlay layer, the overlay layer will attempt to connect to this node. However, no response will be received before a timeout. This may cause high recovery time for neighbor failures.
3.1.2 Overlay Construction & Maintenance

The overlay layer is responsible for constructing and maintaining the overlay structure, upon which application specific communication is carried out. Different applications may require different overlay structures. For example, P2P media streaming applications [15, 75, 59, 22, 79, 50] have used various overlay structures such as trees, multiple trees, meshes and DAGs (direct acyclic graphs), and DHTs such as Chord [74] and Pastry [70] are based on ring-type overlays.

Most P2P applications assume peers will come and go during the lifetime of the application. As a result, overlay construction and maintenance is achieved by handling node joins and departures. For infrastructure based service applications, however, the overlay structure may need to be built from scratch, when the set of application nodes are already up and running.

When an overlay neighbor fails, a straightforward way to repair the failure is to locate an alternative neighbor, which maintains correctness of the overlay structure (e.g., connected and loop free). However, if multiple nodes can fail at the same time, the failure repair mechanism can become very complex. As a result, when a failure occurs, it is also possible that a fresh overlay is constructed, and the previous overlay completely discarded. This approach results in simpler overlay maintenance, although the overlay construction may involve some additional overhead. For both failure repair and on-demand construction, the membership information provided by the lower layer can be especially useful.

3.1.3 Application Specific Processing

The top layer of the OCMA architecture is responsible for data communication and processing that are specific to the application. For example, for media streaming applications, the top layer is responsible for receiving data blocks from neighboring nodes, and assembling them into a media stream. For distributed application status query, the top layer is
responsible for propagating the query down to children nodes (on the overlay tree), and
aggregating the results back. Note the application specific communication is often carried
out on top of the overlay structure, which in turn is determined by the desired application
layer communication patterns.

3.1.4 Discussions

We note the OCMA architecture is very coarse grained. It only provides general guidelines
as to what layers a large distributed application may have and what their functionalities are.
This leaves the application developer with maximum flexibility as to how each layer should be
designed. For example, OCMA does not give a formal specification of the interface between
the layers. Having a fixed interface may be useful, if different layers of an application are
designed by different developers. However, this is unlikely for large distributed applications.
As a result, OCMA decides to give the application developers the freedom of specifying their
interfaces. For example, one application may implement interface such as “return a node
that is less than 30ms away”, while another application may implement interface such as
“return a node that has a playback delay no more than 30 seconds”.

As another example of flexibility, OCMA specifies the functionality that each layer should
provide, but it leaves the detailed design of each layer to the application developer, thus
facilitating innovative design for the layers. The two design techniques we will talk about in
the next section, on-demand overlay construction and control plane services, are examples
of such innovative designs. In contrast, some research work has proposed to use distributed
hashtables as a “common routing substrate” upon which other applications are built. In
fact, a common set of APIs for such routing substrate has been proposed [27]. On the
one hand, such common substrate and API frees the application developers from some of
the low level details such as message routing. On the other hand, it also deprives the
application developers of the flexibility to come up with their own low level designs. It is our
belief that large distributed applications are very different in terms of their communication
characteristics and QoS/performance requirement. Therefore having a common routing layer may be too rigid and may hinder innovation in the design of large distributed applications.

3.2 Novel Techniques for Designing Large Distributed Applications

The OCMA architecture provides application developers the maximum flexibility for innovative design of different application layers. In this section, we describe two novel design techniques that we have explored in designing large distributed applications, namely on-demand overlay construction and control plane services.

3.2.1 On-demand Overlay Construction

Most existing work on large distributed applications has focused on persistent overlay maintenance, which means an overlay is maintained all the time, despite network and node failures. In our Management Overlay Networks (MON) system (to be described in Chapter 5), we have explored on-demand overlay construction, which means no overlay is maintained during normal time. Instead, whenever needed, an overlay can be constructed from scratch and used for a short time. When the overlay is no longer used, it is discarded.

On-demand overlays and persistent overlays are different in several aspects. First, persistent overlays often assume the set of nodes in an application is constantly changing. As a result, the goal is to maintain the correctness of the overlay when new nodes join and existing nodes leave. Such a model is more suited to P2P applications, where peers are autonomous end users. For infrastructure based applications that MON targets, however, nodes do not join or leave the system frequently. When an overlay is needed, the set of nodes to be included are already up and running. Therefore the goal is to build an overlay from scratch with high coverage and good performance.
In addition to application model, a more important difference between persistent and on-demand overlays is their approach to dealing with failures. Persistent overlays attempt to maintain the correctness of an overlay all the time. This could become difficult, since the overlay must be prepared to recover from all possible failures. Note some of these failures may not even be known beforehand. For example, early DHTs were designed with the assumption that any two nodes in an application can directly communicate with each other, as long as they are both alive. However, this was later found to be not true [31, 57]. As a result, new mechanisms are introduced to address such network routing anomalies. While it is always possible to introduce additional mechanisms to deal with newly discovered failures, it would nonetheless make the system complex and difficult to reason about.

In contrast, on-demand overlays focus on building the overlay from scratch and use it for a short time. As a result, it is unlikely that any major failure will happen during the lifetime of an overlay. In case some failures do happen, a new overlay can always be re-built. In this sense, on-demand overlay deals with failures by making them irrelevant. Clearly not all applications can make use of on-demand overlays. However, for the management of a large distributed application, where the goal is to execute some short term status query and control commands, on-demand overlays may provide a simple, scalable and lightweight solution.

On-demand overlay construction and persistent overlay maintenance are complementary to each other. In our MON system, each overlay is built and used for a short time. As a result, there are no failure repair mechanisms. However, it is possible that for some other management commands, the overlay may need to be used for longer time (e.g., monitor metric X for the next 20 minutes). In order to increase the life time of the overlays, we can either make the overlay more resilient by building redundant overlay links, or we can introduce some simple failure repair mechanism. This way the overlay is likely to survive common failure scenarios. In case some unexpected failures cause the overlay to be unusable, we can rebuild another overlay instead. In this sense, there is a continuous spectrum between
no failure repair and repairing all possible failures.

### 3.2.2 Control Plane Services

Existing research on large distributed applications often emphasizes on purely decentralized application design. Purely decentralized design may have good deployability, since there is no need for infrastructure support. However, whenever the application permits, the use of some control plane services may significantly simplify the system design.

A *control plane service* is a small scale distributed system that facilitates large distributed applications in their control plane operations. For example, in a peer-to-peer media streaming application such as the DagStream system (to be described in the next section), peers need to quickly locate good alternative neighbors upon any neighbor failure. Such QoS aware neighbor selection would be difficult to achieve, if the membership information is managed by the unreliable peers themselves. As a result, the DagStream system delegates its membership management to RandPeer [51], a control plane membership service. This greatly simplifies the design of DagStream, since the application is freed from the details of membership management. Each peer only needs to periodically register its information with RandPeer. Whenever needed, they can query RandPeer to find other peers that are potential good neighbors.

In additional to simplifying the application design, a control plane service has the following advantages. First, many applications may have similar control plane operations. If such operations are provided as a service, they can be implemented once and shared by multiple applications. For example, the Chubby locking service [20] from Google allows different application nodes to synchronize with each other. It has been used by different Google applications such as the Google File System [34] and the Bigtable [23]. Second, a control plane service often has much smaller scale than the P2P application itself. As a result, it can be implemented using techniques not possible at large scale. For example, our RandPeer service currently uses DHT for underlying message routing. However, since the RandPeer
service network is likely to be small, we may assume each node has complete membership. This would avoid the multi-hop routing imposed by the DHT. Third, since the control plane service is provided as an infrastructure service, it can act as trusted intermediate between untrusting peers. Such a trusted entity can be especially useful for many P2P issues such as selfishness and accounting.

3.3 DagStream: Locality Aware and Failure Resilient P2P Streaming

In this section, we present the design of an example application that follows the OCMA architecture, namely the DagStream P2P streaming system. The design of another application, the Management Overlay Networks (MON) is presented in Chapter 5, where it is presented as a management tool for large distributed service applications.

Live peer-to-peer (P2P) media streaming has become very popular in the last couple of years. Many P2P streaming systems such as CoolStreaming [79], PPLive [10] and PP-Stream [11] have emerged and attracted hundreds of thousands of Internet users. Despite such success, existing P2P systems are still in their early stage and there are still unresolved issues such as locality awareness.

3.3.1 Existing P2P Streaming Systems

Early P2P streaming systems [39, 15, 75] are designed as alternatives to IP multicast. Therefore, they all attempt to build application level multicast trees. However, a tree structure is unsuited in a P2P environment. First, a tree is vulnerable to node failures. If an interior node in the tree crashes or leaves the system, the whole subtree rooted at the node will be affected. Second, the streaming rate of a peer cannot exceed that of its parent. This means if there is a bottleneck link higher in the tree, the bandwidth of all the downstream peers
Some systems such as CoopNet [59] and SplitStream [22] build multiple trees for streaming. The source media is encoded into multiple layers or descriptions. Each layer or description is streamed on a different tree. The multi-tree solutions can address the bandwidth heterogeneity problem. However, for each tree, the drawbacks of tree-based architectures still exist. For example, the failure of one interior node can still affect the streaming of a whole subtree under that node.

Many recent systems such as CoolStreaming [79], PPLive [10], PRO [67] and Chainsaw [60] build general unstructured meshes for P2P streaming. In such mesh-based overlay networks, each peer can simultaneously download data from multiple other peers, thus when one neighbor fails, a peer can continue streaming from other neighbors while looking for a new neighbor for failure repair.

Although mesh-based overlays provide more flexibility for P2P streaming, one drawback of such systems is that it is difficult to guarantee overall network connectivity. For example, none of the existing systems can prevent the formation of an overlay network as shown in Figure 3.2, where peer $p_2$ (perhaps a powerful peer) is a single point of failure, whose departure would lead to network partitioning (despite the fact that each node has at least two neighbors). Although network partitioning can be detected once it occurs, repairing the partitioning nonetheless takes time and thus affects the streaming quality. The network connectivity problem becomes more serious, if network locality needs to be optimized. This
is because there could be a group of peers that are close to each other, but are far from other peers in the system. If they preferably connect to each other, there could easily be a network partitioning.

Locality awareness is an important performance metric for P2P streaming systems. Without network locality, high bandwidth media streams would go back and forth across the wide area network, which not only increases the delay in receiving the media data, but could easily lead to network congestion.

### 3.3.2 Streaming with DAG Overlays

The drawbacks of tree-based and mesh-based streaming systems have motivated us to design the DagStream system for P2P streaming. As is shown in Figure 3.3, DagStream builds direct acyclic graph (DAG) as the overlay for media streaming. The root of the DAG (s) is the media source, and $p_i$ is the peer in the system. Each node in DagStream has a “level”.

The level of the root is $l_s = 0$, and the level of a peer $p$, whose parent set is $P$, is defined as $l_p = \max_{p_i \in P} (l_{p_i} + 1)$. The level of a peer has a physical meaning. It is the longest path (in terms of overlay hops) from the source. To represent the shortest path from the source, we can define the “min level” of node $p$ as $l'_p = \min_{p_i \in P} (l'_{p_i} + 1)$, where $P$ is $p$’s parent set.

Compared with a tree-based overlay, a DAG allows each peer to stream from multiple parents. Thus it can have better bandwidth utilization and is more resilient to parent failures. Compared with a mesh, a DAG enforces a partial relationship among the nodes (as...
indicated by their levels). As a result, so long as each node \textit{locally} maintains certain number of parents, the whole system will be well connected. In fact, we have the following DAG connectivity property:

\textit{DAG Connectivity Property}: In a directed acyclic graph with one source, if each node (except for the source and its direct children) has at least $k$ parents, then the removal of any $k - 1$ non-source nodes does not cause any remaining nodes to be disconnected from the source.

\textit{Proof}: Suppose there exists a set $P$ of $\leq k - 1$ non-source nodes, and the removal of $P$ causes the network to partition into two components. Consider the component $C$ that does not include the source. There must exist a node $p_i$ in this component, which does not have in degree in the component. Otherwise, if every node has at least one in degree, there would be a cycle in the component. Since $p_i$ doesn’t have in degree in component $C$, in the original DAG, its parents must all be in the set $P$. However, set $P$ only has $\leq k - 1$ nodes, while $p_i$ has at least $k$ parents. This means the removal of $P$ couldn’t have partitioned the network.

The above property means that so long as each peer (except the direct children of the source) maintains at least $k$ parents, the overall network will be well connected. Thus peers can focus on improving their performance such as network locality, without worrying about being disconnected from the source.

The above property also means for any non-source node (except the direct children of the source), there are at least $k$ disjoint paths from the source to the node. This can be proved by applying the Menger’s theorem [17].

The design of DagStream follows the OCMA layered architecture as is shown in Figure 3.4. The middle layer is responsible for building the locality aware DAG overlay for media streaming. The bottom layer manages membership information in a way that allows the upper layer to quickly locate a nearby peer as parent. The top layer is responsible for coordinating the media streaming from multiple parents. In the following, we will briefly describe the design of each of the three layers.
QoS Aware Membership Management

It is well known that P2P applications are highly dynamic. Peers can join and leave the system at any time. As a result, the overlay layer may need to locate alternative parents from time to time, in order to replace a failed parent, or one that has degraded performance. Such “neighbor selection” is implemented by the membership layer. It is clear that when the membership layer returns a node as a potential parent, the node should be alive and have enough bandwidth. For network efficiency, it should also be close to the requesting peer. Implementing such “QoS-aware” membership management for a highly dynamic system is a challenging task.

In our DagStream system, we utilize an external service called RandPeer [51] for membership management. As is shown in Figure 3.5(a), RandPeer is a small scale distributed system that provides membership service to highly dynamic P2P applications. Specifically, each node in a P2P application needs to periodically register itself with the RandPeer ser-
vice. Whenever a peer needs to locate some alternative parent, it sends a lookup request to RandPeer. RandPeer will return a peer that is likely to meet the QoS requirement of the requesting peer. This simplifies the design of P2P applications, because they are freed from the details of storing and organizing the distributed membership information.

Internally, RandPeer uses a logical trie structure to organize membership information, and stores the logic data structure using a distributed hashtable. Figure 3.5(b) shows the membership trie used by DagStream. A trie is basically a tree with its nodes labeled with 0, 1 strings \(^2\). The label of the root is an empty string. If a node has a label \(l\), its left child will have a label \(l0\) and its right neighbor will have a label \(l1\). The nodes in the trie are called “bins”. Membership information of the P2P application is stored in the leaf bins.

To register its membership information with RandPeer, peer \(p_i\) needs to select a random 0, 1 string as its peer id. The peer id determines which leaf bin stores its membership entry. Specifically, the membership entry of peer \(p_i\) is stored in the leaf bin whose label is a prefix of the peer id of \(p_i\). Random ID selection ensures that the membership trie will be roughly balanced. Each leaf bin has a capacity; if there are too many or too few membership entries in a leaf bin, it can be split or merged. The membership entries are soft state. Peers must periodically refresh their registration information. This also allows them to detect any collision in peer ids.

To lookup a candidate parent, peer \(p_i\) can generate a random lookup key and send a lookup message to the leaf bin whose label is a prefix of the lookup key. The leaf bin will return the membership information of the peer whose peer id immediately follows the lookup key.

RandPeer supports QoS aware neighbor selection by clustering peers based on their QoS characteristics. Specifically, peers can map their QoS characteristics (such as geographical location) to a “QoS prefix” in their peer ids. Thus peers with the same QoS prefix will be automatically clustered under the same subtree in the membership trie. When a peer wants

\(^2\)For simplicity, we only discuss binary tries.
to lookup a neighbor with certain desired QoS characteristics, it can generate a lookup key with the specific QoS prefix. The result of such a lookup is likely to be a peer with the desired characteristics. In DagStream, since network locality is our primary goal, we let each peer generate a landmark vector for itself using the landmark binning technique [66] and use the landmark vector as its QoS prefix. When a peer wants to lookup a nearby peer, it uses its own landmark vector as a prefix in the lookup key. Thus the lookup result is likely to be a peer that is nearby.

The membership trie is a logic data structure. In RandPeer, we need a mechanism to store and retrieve the membership bins. This is achieved by building RandPeer on top of the Chord [74] distributed hashtable. Specifically, each node in the RandPeer service is also a DHT node. For a membership bin with label $l$, we use a secure hash function such as SHA1 [12] to map the bin to a DHT node, as is shown in Figure 3.6. That DHT node is responsible for storing the content of the membership bin, and for answering queries about the bin.

**Locality Aware DAG Maintenance**

The middle layer of a DagStream node is responsible for maintaining a locality aware DAG structure. When a node first joins the system, it will use the RandPeer service to locate some initial nodes as parents. Once the node has connected to some parents, it will enter a continuous evolvement phase. This means periodically, the peer will locate a new, potential parent and probe that peer. If the peer can provide better QoS than some existing parent,
the node will connect to the potential parent as a child, and disconnect from some existing parent if necessary. Thus parent failures and performance optimization are handled in the same way.

RandPeer allows DagStream nodes to discover potential good parents much faster, by clustering peers based on their geographical distribution. In addition to this, DagStream has explored two more techniques for parent discovery. First, each node not only maintains a list of its neighbors (parents and children), but also its two-hop neighbors. Whenever a new parent is needed, some nodes in the two-hop neighbor list are tried first. The rationale is that if a neighbor is nearby, its neighbor should also be nearby. The second technique is called parent suggestion. When a node $p_i$ probes $p_j$, it may find that $p_j$ has good QoS characteristics such as delay and bandwidth. However, if the level of $p_j$ is larger than that of $p_i$, $p_j$ cannot be a parent of $p_i$, because that would create loops in the DAG. However, $p_i$ can suggest itself to $p_j$, so that when $p_j$ needs to locate a parent, it will try $p_i$ first.

In addition to parent discovery, a node needs to decide if an existing parent should be replaced by a new parent. This is called the parent selection policy. In DagStream we have explored three policies.

1. **delay only**: A peer always attempts to minimize the delay to its parents. If the delay to a potential parent is smaller than some existing parent, the peer will attempt to connect to the potential parent. When a parent needs to be removed, the parent with the largest delay is always removed.

2. **level only**: A peer always attempts to minimize its level. If the level of a potential parent is smaller than some existing parent, the peer will attempt to connect to the potential parent. When a parent needs to be removed, the parent with the largest level is always removed.

3. **first delay, then level (delay-level)**: A peer will first attempt to minimize the delay to the parents. Once the delay to all parents are within some threshold $d$, the peer
begins to minimize its level. To prevent a peer from over optimizing its level, if the parent that has the largest level is also the one that has the smallest delay, it will not be replaced.

**Multi-parent, Receiver-driven Streaming**

On top of the locality aware DAG overlay, peers can stream media data using a multi-parent, receiver-driven approach. As is shown in Figure 3.7, at any time, a node will have a set of media blocks buffered in its memory. Whenever a node receives a new block, it will announce this to all of its children. As a result, each node knows which blocks are currently available at each parent. The node will then make a scheduling decision and request different blocks from different parents. For example, in Figure 3.7, peer $p$ may decide to request block 5 from parent $q_1$, block 7 from $q_2$, and block 8 from $q_3$. Block scheduling is itself an interesting problem. In DagStream, we have focused on building the locality aware DAG overlay. We leave the block scheduling algorithm as our future work.

### 3.3.3 Experimental Results

We have fully implemented our DagStream protocol. In this subsection, we mainly present our simulation results to show (1) the ability of DagStream to build locality aware DAGs, as well as the impact of different parent discovery techniques and parent selection policies; (2) the ability of DagStream to deliver good streaming quality, by allowing peers to select additional parents, so long as their streaming quality can be further improved. We also
present some results from our experiment on the PlanetLab [62] testbed.

Since we want peers to preferably connect to nearby parents with small levels, we evaluate the protocols using two metrics. The first is the average parent-child delay, which indicates on average how far a peer is from its parents. The second metric is the max level and max “min level” of a DAG overlay. The max level of a DAG overlay is the longest overlay path from the source \( s \) to any node. The max min level is the worst case shortest overlay path from \( s \) to any node. Alternatively, if the max min level of an overlay is \( l \), it means every peer has at least one overlay path from the source that has a length \( \leq l \).

In all the following simulations, we use the BRITE [56] topology generator to generate a two level hierarchical network topology. We then randomly select a subset of the nodes as peers in DagStream. We use all-pairs shortest path algorithm to compute the end-to-end delay between the peers, and normalize the delay to a maximum of 100 ms. Unless otherwise specified, the minimum number of parents for each node is 2. The maximum number of parents is \( P_{\text{max}} = 5 \), and the maximum number of children is \( C_{\text{max}} = 5 \). By default the delay-level parent selection policy is used, and the delay threshold \( d \) is set to 25 ms. Each node sends a probe message every 10 seconds. When using RandPeer, each peer generates a 3-bit landmark vector [66] as its QoS prefix. Each bit encodes the delay of the peer to one landmark node.

**Impact of Parent Discovery Techniques**

Figure 3.8 illustrates the impact of different parent discovery techniques on building locality aware DAGs. For this experiment, each time we select a specific number of nodes from the network topology as DagStream peers. Initially no peer is connected to the source. We let the peers join the system and evolve the network for 1000 seconds and measure the average parent-child delay of the DAG. Each experiment is repeated 200 times and the averages are presented. During the evolvement peers use different techniques for parent discoveries, “random” means peers select potential parents purely randomly from the entire system,
Figure 3.8: Parent-child delay of different parent discovery techniques.

and “DagStream” refers to the combination of using RandPeer, two hop neighbors and the parent suggestion techniques.

Figure 3.8(a) shows that first, compared with random neighbor selection, the QoS awareness of RandPeer can significantly improve the locality of the resulting overlay, especially for large networks. For a 1000 peer network, RandPeer alone achieves more than 20% improvement compared with random neighbor selection; second, the use of two hop neighbors can greatly improve the parent-child delay. This is because the two hop neighbors of a peer are also likely to be close neighbors of the peer. The parent suggestion mechanism also improves the locality of the overlay. But its effect is less significant.

Figure 3.8(b) compares the parent-child delay with that of the minimum spanning tree (MST). We can see that the delay ratio of DagStream increases as the network size increases. For a 1000 peer network, the average parent-child delay of DagStream is a little more than twice that of the MST. This is because in DagStream, each peer has at least two parents, thus the DAG overlay has twice as many links as the MST. Also we limit the maximum out degree of each node to 5, while the MST has no such constraints. Finally, DagStream attempts to optimize level when the parent-child delay is within some threshold, while MST
makes no attempt to reduce its level (height).

Figure 3.9 compares the level of MST and DagStream. The figure shows that when the network size grows, the height of the MST grows quickly. And for large networks, even the max level of DagStream is much smaller than the level of the MST. The max min level of DagStream grows slowly with the network size. For a 1000 peer system, the max min level of DagStream is 10. This is only 3 hops larger than the random parent discovery, even though the delay of DagStream at each hop is only half that of the random parent discovery.

**Effect of Parent Selection Policies**

Figure 3.10 shows the effect of different parent selection policies as introduced in the previous subsection. Figure 3.10(a) is the average parent-child delay of the policies. As we can see, the *level only* policy always tries to minimize the level of a peer, without considering the locality of the parents. As a result, the average parent-child delay is always about 50ms, which is the average delay between random peers. The *delay only* policy always tries to minimize the delay to the parents. However, the achieved delay is worse than that of the *delay-level* policy. For larger networks, the average parent-child delay of the *delay-level* policy is about 14% smaller than that of the *delay only* policy. The reason is that the *delay-level* policy tries to reduce the level a peer once its parents are within the delay threshold $d$. When the level
of a peer decreases, it is eligible as a parent for more peers. This increases the chances of other peers to locate good parents.

Figure 3.10(b) shows the max level and max min level of the different policies. Not surprisingly, the level only policy achieves the smallest max level and max min level. However, the max min level of the other two policies are only 3 or 4 hops larger than that of the level only policy. The figure shows that the max min level of the delay only and delay-level policies are the same. This is because we rounded off the average values to integers.

The delay-level policy uses a delay threshold $d$ to decide when to switch from delay optimization to level optimization. Thus $d$ may have an impact on the performance of the policy. Figure 3.11(a) shows the effect of the delay threshold on the parent-child delay. Each line is for a different number of peers. We can see generally when $d$ is larger, peers will focus less on improving the parent delay. As a result, the achieved parent-child delay is larger. Thus using smaller $d$ may improve the parent-child delay. However, when $d$ is too small, the achieved delay will actually increase, this is due to the same reason that caused the delay only policy to perform poorly. When peers do not attempt to reduce their level, it is less likely for other peers to choose them as parents. Figure 3.11(b) shows the max level for different $d$. Indeed when $d$ is large, peers focus more on improving their level, thus the max
level will decrease. The effect of $d$ on the max min level is similar, although not significant.

**Improve Streaming Quality**

DagStream focuses on improving the network locality of the underlying overlay network, and allows peers improve their streaming quality by selecting additional parents as needed. Figure 3.12 shows this approach can indeed deliver good streaming quality. For this experiment, we first generate the uplink and downlink bandwidth of the peers using a distribution similar to those reported in [72]. 20% of the peers are low bandwidth peers. Their downlink bandwidth is distributed in the range [384kbps, 1Mbps], and uplink bandwidth in the range [128kbps, 384kbps]. 50% of the peers are with medium bandwidth. Their downlink and uplink bandwidth ranges are [1Mbps, 6Mbps] and [384kbps, 1Mbps]. Finally, 30% of the peers are with high bandwidth. Their downlink bandwidth is in the range [10Mbps, 50Mbps], and their uplink bandwidth is equal to their downlink bandwidth. All the distributions are uniform. While our bandwidth distribution may not exactly match a particular P2P system, it nonetheless introduces the heterogeneity that is typical in a P2P environment. The number of peers is 1000 for this experiment.

In the experiment, peers first attempt to connect to $k$ nearby parents. They then compute
the aggregate streaming rate provided by their parents. If the aggregate rate is smaller than the target rate (the rate of the original video), and their downlink bandwidth has not been fully utilized, they will attempt to connect more parents, which are discovered and probed in the usual way. We let the system evolve for 1000 seconds, and count the number of peers with satisfying streaming quality. A peer is satisfied with its quality, if the aggregate rate provided by its parents is larger than the target rate or its own downlink bandwidth. For simplicity, we assume the uplink bandwidth provided by a peer is equally shared by its children.

Figure 3.12: Streaming quality delivered by DagStream.
Figure 3.12(a) shows the number of peers that have good streaming quality (are satisfied) as a function of the maximum number of parents $P_{\text{max}}$ that a peer is allowed to connect to. The maximum number of children that a peer can accept ($C_{\text{max}}$) is fixed to 12. We can see that when $P_{\text{max}}$ is limited to 5, almost every peer will have good streaming quality when the target rate is 200kbps. But only about 88% peers have good streaming quality when the target rate increases to 700kbps. By relaxing $P_{\text{max}}$, however, peers can aggregate bandwidth from more parents and thus more peers will have good streaming quality. When $P_{\text{max}}$ is 9, about 99% of the peers are satisfied even for a target rate of 700kbps. Note that in our bandwidth distribution, only less than 55% peers have uplink bandwidth that is $\geq$ 700kbps.

Having $P_{\text{max}} = 9$ might seem to be too large. However, Figure 3.12(b) shows that even though $P_{\text{max}}$ is set to a large value, the average number of parents for a peer is still small. For example, even when the target rate is 700kbps and $P_{\text{max}}$ is 9, on average a peer has less than 3.1 parents. In fact, when $P_{\text{max}}$ increases, the average number of parents may decrease slightly. The reason is that when $P_{\text{max}}$ is small, a peer may connect to $P_{\text{max}}$ parents that have low uplink bandwidth. In this case the peer is not allowed to try more parents, and it will not disconnect from existing parents, because this will further reduce its streaming rate. When $P_{\text{max}}$ is large, however, the peer has the opportunity to try more parents, and as a result discover parents with large uplink bandwidth. At this time the peer is satisfied with its quality and can disconnect some parents with low uplink bandwidth. This result shows to allow peers to explore more parents when needed, $P_{\text{max}}$ should probably not be set too small. Allowing a peer to connect to $P_{\text{max}}$ parents doesn’t necessarily mean it will always maintain $P_{\text{max}}$ parents.

One goal of DagStream is to let peers stream preferably from nearby parents, and to connect to remote parents only when necessary. Figure 3.12(c) shows DagStream achieves this goal. When the target rate is 200kbps, on average the delay of a peer from its parents is only a little more than 25 ms. When the target rate increases, the average parent-child delay increases, because peers have to connect to more parents, which might be farther
away. However, even when the target rate is 700kbps, the average parent-child delay is still less than 40ms, which is about 20% less than the average delay between any two peers. An interesting thing is that when $P_{\text{max}}$ increases, the average parent-child delay actually decreases. The reason is the same as for the average number of parents. The initial parents of a peer may not be good, both in terms of uplink bandwidth and delay. By allowing peers to try more parents, they have better chance to discover parents that not only have large uplink bandwidth, but also are nearby.

**Performance on PlanetLab**

We have experimented with DagStream on about 228 PlanetLab [62] nodes that are distributed world wide. Each time we start DagStream on all nodes at the same time, and let the nodes join the system and evolve the overlay structure. After the system has been started for 60 seconds, we begin to measure performance of the DAG overlay every 20 seconds. This is done by a local client sending UDP packets to probe all nodes in the system. We repeat the experiment 10 times and show the average results. Since the network is relatively small, we use gossip as for membership management. Each node initially only knows about the source (planetlab2.cs.uiuc.edu). Information about other nodes is learned by contacting...
the source. Figure 3.13(a) shows the number of nodes that joined the system over time. We can see most nodes joined the system within 2 minutes. Note each peer tries a candidate parent every 10 seconds. If the candidate is full, the peer will wait for 10 seconds and try again.

Figure 3.13(b) shows the average inter-node delay (the delay between two random nodes) and the parent-child delay for different delay threshold $d$. The inter-node delay is measured among those nodes that have already joined the system. Since each node joins the system by contacting the source, the nodes that are closer to the source (and thus closer to each other) joined the system first. As a result, the inter-node delay is smaller at first. However, as more nodes join the system, the inter-node delay increases to about 80ms (because many of the nodes are in Asia and Europe). For the parent-child delay, initially it is relatively large. However, as nodes discover and switch to better parents over time, the delay quickly decreases to about 30ms, which is more than 60% less than the inter-node delay. The figure shows that using a delay threshold of $d = 10\text{ms}$ achieves smaller average parent-child delay than $d = 20$ and $d = 30$. However, the difference is not too much.
Chapter 4

A Reusable Framework for Implementing Service Applications

In this chapter, we present PPF (Protocol Plug-in Framework), a reusable C++ framework for implementing large distributed applications. PPF is extracted from our development of multiple large distributed applications including MON and DagStream. Thus even though our focus is on service applications, the same framework can be used for P2P applications as well. PPF supports the development of event driven, single thread applications. Applications developed with PPF can automatically run in simulation and real world mode. In simulation mode, PPF provides support for simulating network topology, network and node failures, and statistics report. In this chapter, we first present an overview of PPF. We then provide some details on how PPF supports both simulation and real world execution mode.

4.1 PPF Overview

Figure 4.1 shows the PPF framework. It consists of an event manager, a time manager, a socket manager, one or more peer nodes, and one or more protocol modules on each peer node. The protocol module is implemented by the application developer. All other components are provided by PPF and can be directly reused.

The event manager is the central component of the application. It consists of an event queue and a dispatcher. The event queue is a priority queue that keeps the events sorted according to their firing time. The events will be dispatched to their corresponding handlers in the order of their firing time. The time manager provides time simulation. Whenever some component in the framework needs to obtain the current time, it will go through the
Figure 4.1: Components of the PPF framework

time manager. The time manager can return either the real time, or some virtual time, depending on the execution mode.

The protocol module is the main component of the application. It is implemented by the application developer (by extending an abstract protocol module provided by the PPF framework). A protocol module mainly provides event handlers for timer and network events. For example, when a periodic timer event fires, the protocol module may send a gossip message to some random peer. When a gossip message is received, the protocol module may send an acknowledgement message back.

The peer node is the component that implements network simulation. It provides the network communication interface that the protocol module can use to send network messages. In the simulation mode, such messages are scheduled as events. In the real world mode, the messages are passed to the socket manager, which implements asynchronous network communication. Whenever a network message is received by the socket manager, it will pass the message to the peer node, which then calls the message handler of the protocol module to handle the message.

Note an application may have multiple protocol modules. For example, Figure 4.2 shows an example application designed according to the OCMA architecture. Each application node thus consists of the membership, overlay and application specific layers, together with other components from the PPF framework. Each layer can access the communication
interface that the peer node provides, and each can have its own message handlers.

![Diagram of components of an example application with three layers](image)

**Figure 4.2: Components of an example application with three layers**

PPF uses an event driven architecture. The whole system has only one thread. When the system starts, it creates and initializes various components. During the initialization, each component can schedule some timer events to be executed. After the initialization, the system enters the dispatcher loop of the event manager. Assume the time and network are not simulated, the dispatcher loop will look like that shown in Figure 4.3.

```cpp
for(;;){
    t1 = event_queue.GetNextEventTime();
    t0 = time_manager.GetCurTime();
    while (t1 <= t0){
        Event* event = event_queue.DeQueue();
        event->owner->HandleEvent(event);
        t1 = event_queue.GetNextEventTime();
    }
    //here t1 is > t0
    sock_manager.HandleSockEvents(t1 - t0);
}
```

**Figure 4.3: Main dispatcher loop for real time, real network**

As the figure shows, the dispatcher repeatedly removes the next event and calls its handler, if the event firing time has passed. Otherwise the dispatcher calls the socket manager to process socket events. The socket manager uses the `select()` system call to monitor and process the system events (e.g., the receipt of a network message).
for(;;){
    t1= event_queue.GetNextEventTime();
    time_manager.SetCurTime(t1);
    Event* event = event_queue.DeQueue();
    event->owner->HandleEvent(event);
}

Figure 4.4: Main dispatcher loop for simulated time and network

Figure 4.4 shows the dispatcher loop if both time and network are simulated ¹. Since network messages are also delivered as events in the simulation mode, the socket manager is not used, and the simulation time is advanced as soon as one event is finished.

4.2 Time and Network Simulation

PPF supports the separate simulation of time and network. As a result, applications developed with PPF can execute in three modes: (1) both time and network are simulated. In this mode, the application execution is repeatable, thus it is useful for application debugging. (2) time is real, but network is simulated. This mode allows the application developer to simulate large system and at the same time interact with the system from the console. (3) both time and network are real. This allows the application to run in a real world, distributed environment, communicating with each other via network messages ². Below we describe how PPF implements time and network simulation.

The simulation of time is in fact fairly simple. As described in the previous section, the time manager provides an interface to get the current time. Whenever a component needs to get the current time, it will call this interface. If time is simulated, the simulated time is returned. Otherwise, the real system time is returned.

For the simulated time, we need to advance the time appropriately. In PPF, we assume

¹In PPF, there is only one dispatch loop. Here we show the dispatch loop for real world and simulation mode separately, just for clarity purpose.

²The fourth execution mode, where time is simulated but network is real, is not useful since time cannot progress like in Figure 4.4 without regard to the (real) time taken for sending/receiving network packets.
when time is simulated, the network is also simulated. Therefore, whenever the event dis-
patcher finishes dispatching one event, it informs the time manager to advance the system
time to the next event time (as shown in the line 3 of Figure 4.4). This way, the simulated
time can progress much faster than the real time. For example, instead of sleeping for 10
seconds before sending the next gossip message, we can advance the simulated time by 10
seconds and immediately send the gossip message.

Network simulation allows application developers to simulate large number of application
nodes without using much system resource (e.g., sockets), and to introduce network failures.
Thus it is extremely useful for debugging network protocols at large scale. In PPF the
peer node provides a set of network APIs that are similar to the standard socket APIs.
These APIs isolate protocol modules from the real network, and provide the opportunity for
network simulation.

We first look at UDP simulation. The peer node provides a simple \texttt{udp\_send()} interface.
Any protocol module can use this interface to send an UDP message to any other peer node.
To identify a receiving peer node, a sending peer node must specify the receiver’s IP, port
and a numerical peer ID. If the network is not simulated, a UDP message is sent to the IP
and port of the receiving peer node. Otherwise, a “message event” is scheduled. When the
event fires, the message is delivered to the receiving peer node, which is identified by its ID.
Each protocol module provides a callback function \texttt{HandleUDPMessage()}, and registers its
message types with the peer node. When a peer node receives a message, it delivers the
message to the protocol module determined by the message type (note on one peer node,
there could be multiple protocol modules. The peer node will de-multiplex received messages
among the protocol modules).

The simulation of TCP connection is a little more complicated. First, TCP is connection
oriented. To faithfully simulate TCP connection, we need to explicitly simulate the connec-
tion establishment process. This will allow the application code to remain unchanged when
we switch between simulation and real world mode. In PPF, we provide several APIs that re-
semble the real TCP connection establishment. These include `create_tcp_server_socket()` and `create_tcp_socket()`. The former creates a TCP server socket and binds to a specified port, while the second creates a client socket and connects to a server socket. Second, asynchronous network programming with TCP is difficult, because the sender may send a large message, which is divided into multiple IP packets and received by the receiver via multiple `recv()` system calls. To simplify the asynchronous network programming, we provide a message level simulation. Specifically, the peer node provides a `tcp_send()` API, and the protocol module needs to provide a `HandleTCPMessage()`. When the network is simulated, each TCP message (the data sent in one `tcp_send()`) is delivered as an event, similar to the UDP case. However, if the network is not simulated, the socket manager will perform asynchronous network I/O. Once a TCP message is fully received, it will be passed to the peer node, which then de-multiplexes it and delivers it to the right protocol module.

For network simulation, the peer node can make use of a simulated network topology in the form of a delay matrix. The delay matrix specifies the network delay between any pair of application nodes. When a protocol module sends a message in the simulated mode, the peer node can look up the delay \( d \) between the sender and receiver, and schedule the message event to fire in \( d \) time units. The application developer can also specify a loss matrix, which specifies the probability that a message will be lost between a pair of nodes.

4.3 Support for Application Evaluation and Management

Since an application developed with PPF is event driven, it is very straightforward to plug in some additional protocol modules. For example, PPF has implemented a generic failure injector. When it is initialized, the failure injector schedules a timer event. When the

---

3In order to support repeatable simulation, PPF needs to use single thread. Otherwise thread scheduling (either at user level or kernel level) will introduce non-determinism. To handle concurrent network I/O with single thread, the I/O has to be asynchronous.
timer event fires, the failure injector can introduce massive node failures, random node failure and recovery, and other application specific failure modes. To support node failure injection, the abstract protocol module provides two interfaces \texttt{GoOn()} and \texttt{GoOff()}. The derived protocol module should implement these two interfaces to mimic node failures and recoveries (e.g., release and reallocate resources). Application developers can also implement performance reporters that can periodically inspect the whole system and compute the desired performance metrics.
Chapter 5

Management Overlay Networks for Service Applications

In this chapter, we present the design and implementation of Management Overlay Networks (MON), a simple and lightweight tool for managing large distributed service applications. We first provide an overview of MON. Next, we present the detailed design of MON, including its membership management and overlay construction algorithms. Finally, we discuss the management support of MON and provide some evaluation results. MON is not just a management tool. It is a large distributed application by itself and it is designed according to the OCMA architecture as described in Chapter 3.

5.1 MON Overview

When a service application is running in a wide area environment, the application developer may not have good visibility into the application status. For example, the application may be running correctly on 490 out of 500 nodes, but may have encountered unrecoverable failures on the remaining 10 nodes. Some of these failures are due to programming errors, but others may be caused by the environment (e.g., an application unable to write to the disk, due to some other application that has used up the disk space). In order to detect and diagnose such problems, we need to monitor detailed information about the application processes. For example, if a process is using too much memory (compared with the same process on other computers), it might be an indication of memory leak. Also, many applications would output valuable debugging information to log files. If we can simultaneously query the distributed log files generated by an application, we may be able to quickly find out which application
nodes, if any, have encountered certain error conditions.

Existing management tools for distributed systems have focused on continuously monitoring a small set of predefined metrics such as CPU load and free memory. Such continuous monitoring is insufficient for application level management, because there may be too many application metrics that an application developer may want to monitor. As a result, we believe dynamic query is more useful. Instead of proactively collecting all the metrics to a central location, whenever needed, a query is pushed down to all application nodes and the results returned. Also, when an application metric is queried, it may be desirable to return the aggregate state such as average and top K, rather than the raw data, due to the large volume of such raw data.

The above idea is illustrated in the left part of Figure 5.1. To effectively manage a service application running in a wide area environment, we need a mechanism to push the query to all application nodes, and to return the aggregate results.

To support dynamic, aggregate query and control, one way is to use a centralized architecture. However, this may have scalability problems when the application size increases. An alternative is to build a distributed overlay (e.g., a tree overlay) and use it to manage the service application throughout its life time. However, service applications often need to run for a long time. Maintaining a tree overlay for a long time not only increases maintenance overhead, but also makes the system more complex since the management overlay

Figure 5.1: Management overlay network
itself needs to recover from all possible failures.

This has motivated us to take an on-demand approach. Specifically, during normal time, MON does not maintain any fixed overlay. Instead, whenever the application developer needs to execute one or more management commands (query or control), MON can dynamically set up a control plane overlay to execute the commands. Once the commands are finished, the overlay is removed. Since no overlay is maintained for a long time, there is no need for complex failure repairs. The right part of Figure 5.1 illustrates this approach. Note the management overlay is different from the application overlay, this makes MON generic since it can manage different applications.

To build management overlays on-demand, we deploy a MON daemon on each computer where the application is running. The design of MON follows the OCMA architecture. As is shown in Figure 5.2, each MON daemon consists of three layers. The bottom layer is for membership management. The middle layer is for on-demand overlay construction. And the top layer is for query and control command execution. During normal time, the membership layer exchanges messages with each other to maintain up-to-date membership information. Whenever some query commands need to be executed, the middle layer constructs an overlay on-demand, using the membership information. The top layer then propagates the query to each node, executes the query locally on each node, and returns the aggregate result. The local execution of the query may involve querying the operating system, or some management interface (as shown in Figure 5.2) provided by the application being managed. Note the managed application may itself be implemented according to some layered architecture (such as the OCMA architecture in Chapter 3).

5.2 Gossip Based Membership Management

In this section, we describe the design of the membership layer of the MON system. MON is deployed in the same environment as the application being managed. Therefore, it is also
affected by the node crashes/recoveries and network failures. The goal of the membership layer is to maintain up-to-date information about some other nodes (MON daemons) in the system. Such membership information can be used to build on-demand overlays by the middle layer.

At the membership layer, each node (MON daemon) maintains membership information for $m$ random nodes in the system. This is called its random membership view. Each membership entry in the view contains the following information: (1) $peer\_id$, which is a logic ID for the peer node; (2) $peer\_addr$ and $peer\_port$, the IP address and port number of the peer; and (3) $birth\_time$, which indicates the freshness of the membership entry. More specifically, it means the corresponding peer was alive at $birth\_time$. If the current local time is $current\_time$, the age of the entry is $current\_time - birth\_time$. We will discuss this in more details later.

5.2.1 Membership Gossip Protocol

The gossip style membership protocol is shown in Figure 5.4. As the figure shows, each node will periodically select a random node from its membership view as target, and send a ping message to the target. The message also includes $num\_entries$ entries from the local membership view. When the target node receives the message, it merges the membership entries with its own membership view, and sends a pong message back, which also includes $num\_entries$ entries from its membership view.
One goal of the membership protocol is to maintain up-to-date membership information. This means if a node has failed, it should be removed from the membership view as soon as possible. Traditionally, failure detection is often achieved using heartbeat messages. Specifically, two nodes $p_i$ and $p_j$ will exchange heartbeat messages periodically. If $p_i$ does not receive heartbeats from $p_j$ for $c$ consecutive rounds, it will declare $p_j$ as failed. Gossip style protocol cannot use heartbeats for failure detection because nodes do not have fixed neighbors. Therefore, we use birth time to estimate the freshness of a membership entry.

Figure 5.3 illustrates how the birth time is maintained. Suppose at some point in time, the membership views of nodes $A$ and $B$ are as shown in Figure 5.3(a). Note $A$ and $B$ may have different local time, and the birth time of the membership entries are relative to the local time. Suppose at this time, $A$ sends a ping message to $B$. If entries $C$ and $E$ are included in the message, $A$ will convert their birth time to their ages, which are 15 and 30, respectively. When $B$ receives the message and merges the entries, its membership view will look like that in Figure 5.3(b) (ignoring the network transport delay for the message). $B$ has learned two new entries $A$ and $E$. Note their birth time is relative to $B$’s local time. The birth time of $C$ is not updated, since $B$ has more recent liveness estimate for it. However, if $B$ sends a pong back to $A$, and includes $C$ in the message, $A$ will update its entry $C$ to have a birth time of 73 (i.e., age = 7). Note if the size of the membership view is $m = 3$, $B$ will purge two entries, e.g., $D$ and $E$. Suppose 10 seconds later, $B$ gossips with $C$ and includes the entry for $A$, $C$ will learn that $A$ was alive at least 10 seconds ago, even though $C$ may have never exchanged message with $A$.

5.2.2 Analysis of the Membership Protocol

If node $p_i$ has a membership entry for $p_j$, and the corresponding age is $t$, it means $p_j$ must be alive $t$ seconds ago, because it sent out a message at that time. Suppose the life time of the nodes are exponentially distributed with a mean of $T_0$ seconds, then the probability that $p_j$ has failed in the last $t$ seconds is $1 - e^{-t/T_0}$. Therefore, the smaller the age of an entry, the
more likely the corresponding node is still alive. Such *probabilistic notion* of node failure is different from heartbeat-based failure detection, which would classify a peer as either alive or failed.

In our membership protocol, if a node \( p_i \) is alive, it will periodically send out gossip messages. The receivers will then update the *birth time* for \( p_i \) to the current time (line 3 of the *MergeView*() function in Figure 5.4). Also, when nodes exchange gossip messages, they will update the *birth time* of their membership entries (line 6 of the *MergeEntry*() function in Figure 5.4). As a result, if \( p_i \) is alive, other nodes will have a small age for it (if \( p_i \) is in their membership view). On the other hand, if \( p_i \) has failed, its age will not be updated. Eventually other nodes will remove it since its age is larger than some threshold.

One may wonder how such gossip-based membership protocol compares with traditional heartbeat-based protocols. We do some analytical comparison of the average age of a membership entry for heartbeat-based membership management and our gossip-based membership management. The smaller the average age, the more up-to-date information a node has about other nodes.

For the heartbeat-based membership management, suppose a node \( p_i \) maintains \( m \) fixed neighbors, and it sends a heartbeat message to one of the neighbors (in round robin fashion)
Random Membership Maintenance Protocol

Periodically
1. target = SelectRandomEntry(1)
2. list = SelectView(num_entries)
3. SendMessage(ping, target, list)

Upon receipt of ping message from sender
1. let list be the membership entries in the message
2. MergeView(sender, list)
3. list2 = SelectView(num_entries)
4. SendMessage(pong, sender, list2)

Upon receipt of pong message from sender
1. let list be the membership entries in the message
2. MergeView(sender, list)

function SelectView(num_entries)
1. list = SelectRandomEntry(num_entries)
2. for each entry e in list
3. e:age = current_time − e.birth_time
4. end for;
5. return list

function MergeView(sender, list)
1. e = FindEntry(sender)
2. if e == null then e = CreateEntry(sender)
3. e.birth_time = current_time
4. MergeEntry(e)
5. for each entry e in list
6. e.birth_time = current_time − e.age
7. MergeEntry(e)
8. end for

function MergeEntry(e)
1. e' = FindEntry(e.peer_id)
2. if e' == null
3. InsertEntry(e)
4. if there are > m entries then remove a random entry
5. else if e'.birth_time < e.birth_time
6. e'.birth_time = e.birth_time
7. end if

Figure 5.4: Membership view maintenance protocol.
every \( T \) seconds. If we look at one neighbor \( p_j \), \( p_i \) will send a heartbeat to \( p_j \) every \( m \cdot T \) seconds. Because \( p_j \) is also sending heartbeat messages to its neighbors, in the best case, every \( m \cdot T/2 \) seconds, \( p_i \) exchanges message with \( p_j \)\(^1\). Therefore, the average age for a membership entry is at least

\[
A = m \cdot T/4
\]  

(5.1)

For the gossip-based protocol, assume in the steady state, the average age of \( p_i \)’s membership entries is \( A' \). In every \( T \) second period (round), \( p_i \) expects to receive two gossip messages, one pong and one ping. Suppose each message contains no entries other than the sender (i.e., \( \text{num\_entries} \) is 0 in the function \( \text{SelectView()} \) in Figure 5.4), \( p_i \) will learn about two new entries. Thus in the next round, \( m - 2 \) entries will have an average age of \( A' + T \) in the worst case, while two entries will have an age of at most \( T \). Therefore we have

\[
A' = \frac{(m-2) \cdot (A' + T) + 2T}{m}, \text{ which means}
\]

\[
A' = \frac{m \cdot T}{2}.
\]  

(5.3)

Note the above analysis assumes \( \text{num\_entries} \) to be 0. When each gossip message contains multiple membership entries, these entries may update the membership at the receiver. Therefore as \( \text{num\_entries} \) increases, we expect the average age of the membership entries to slightly decrease.

In the above, we consider the average age of the membership entries in a node \( p_i \)’s membership view. If we consider all the nodes that have \( p_i \) in their membership view, and their age estimate for \( p_i \), the same analysis also applies. Therefore, if a node is alive, it is likely other nodes will have an age estimate of \( A' \) for it.

The \( \text{SelectView()} \) function in Figure 5.4 selects random membership entries to be included in the gossip messages. This may be undesirable since even after a node has crashed,

\(^1\)The worst case is, \( p_i \) and \( p_j \) always send heartbeat message to each other at exactly the same time. As a time, they will exchange message every \( T \) seconds.
its membership entry may still be gossiped around by other nodes. Therefore, we have investigated two different view selection algorithms. The first is called *Greedy*, it means the membership entries with the smallest ages are selected. The second is called *Hybrid*, which means a node will divide its membership entries into the younger half and the older half. The entries to be included in the gossip message are randomly selected from the younger half of the membership view.

For the *Hybrid* view selection algorithm, suppose each message contains `num_entries` entries, and the two messages that a node receives in one round have distinct entries, the average age $A'$ will satisfy the following (here we assume if some nodes are randomly selected from the younger half, their average age is $A'_0$):

$$A' = \frac{(m - 2 - 2 \cdot \text{num\_entries}) \cdot (A' + T) + 2T + 2 \cdot \text{num\_entries} \cdot (\frac{A'_0}{2} + T)}{m}, \text{ or } (5.4)$$

$$A' = \frac{(m - \text{num\_entries}) \cdot T}{2 + \text{num\_entries}}. \quad (5.5)$$

Figure 5.5 shows simulation results for the three view selection algorithms (*Random*, *Greedy* and *Hybrid*). The system has $N = 1000$ nodes, each node keeps $m = 40$ entries in its membership view. Each node gossips every $T = 10$ seconds. The figure shows that when `num_entries` is 0, the average ages for different algorithms are the same, i.e., the ages are all
In degree distribution, for random view selection

In degree distribution, for hybrid view selection

In degree distribution, for greedy view selection

(a) Random view selection  (b) Hybrid view selection  (c) Greedy view selection

Figure 5.6: Comparison of in-degree distribution

close to $\frac{mT}{2} = 200$ seconds (as predicted by Equation(5.3)). When $num\_entries$ increases, the average age for all algorithms decreases, due to the additional membership information included in the gossip messages. The average age for Greedy and Hybrid algorithms decreases much faster than Random, because they propagate more up-to-date membership information. When $num\_entries = 1, 3$ and 5, the average age for Hybrid algorithm is about 130, 80 and 60, which is close to that predicted by Equation(5.5).

For comparison, we have simulated a system with fixed neighbors. Using the same message overhead (i.e., $T = 10$ and $m = 40$), the average age of the membership entries is about 130 seconds, about 30% higher than that predicted in Equation(5.1) (note the result in the equation is the best case).

If we use random view selection, it is likely that the membership view of any node represents a uniform sampling of the whole system. However, Greedy and Hybrid may affect such property, even though they may reduce the average membership age. We have evaluated one aspect of the membership distribution, namely the in-degree distribution of the nodes. We find that the in-degree distribution of Hybrid is very similar to that of Random, while the distribution for Greedy is significantly different from the other two algorithms (as shown in Figure 5.6). Therefore, in practice we may use the Hybrid view selection algorithm, which balances the randomness of view selection and the desire to gossip more up-to-date
5.3 On-demand Overlay Construction

In this section we describe how MON builds overlay networks on-demand. Since an overlay tree is ideal for distributed status query and control, we will focus on building on-demand trees. To provide redundancy in the overlay, we also describe how to create on-demand DAGs (direct acyclic graphs).

5.3.1 General Algorithm

We first describe the general algorithm that we use for on-demand tree construction. The algorithm is shown in Figure 5.7. Specifically, whenever the user wants to build an on-demand overlay network, starting from some initiator node, each node will select \( k \) nodes (called the children nodes) and propagate a tree construction message (session message) to each of them. Each node, when it receives the message for the first time, will repeat the process and further propagate the message to \( k \) other nodes. When a node receives a session message for the first time, it will respond with a session.ok message. Otherwise it will respond with a prune message. When a node receives session.ok from all of its children, it will send a session.ok to its parent. Once the user receives a session.ok message, the overlay tree has been constructed.

To evaluate the construction algorithms, we use two performance metrics: coverage and response time. coverage means the percentage of live nodes that are included in an on-demand overlay, and response time means after a management command is issued, how long does it take to get the result back. It can be seen that the performance of the algorithm depends on how each node propagates the session message (i.e., which nodes are returned by the SelectChildren() function). Below we describe three specific algorithms: randk, leafset+RF and tree+RF.
The General Tree Construction Algorithm

Initiator: upon request from user
1. \texttt{children\_list} = \texttt{SelectChildren}(k)
2. for each \texttt{child} in \texttt{children\_list} do; send \texttt{session} message to \texttt{child}; end for
3. set current state to not ready
4. start retransmit timer

Non-initiator: upon receipt of \texttt{session} message from \texttt{sender}
1. if the message has been received before then
2. send \texttt{prune} back
3. return
4. end if
5. set \texttt{parent} = \texttt{sender}
6. \texttt{children\_list} = \texttt{SelectChildren}(k)
7. for each \texttt{child} in \texttt{children\_list} do; send \texttt{session} message to \texttt{child}; end for
8. set current state to not ready
9. start retransmit timer

Upon receipt of \texttt{prune} message from \texttt{sender}
1. remove \texttt{sender} from \texttt{children\_list}
2. if \texttt{session\_ok} has been received from all existing children
3. set current state to ready
4. send \texttt{session\_ok} to \texttt{parent}
5. end if

Upon receipt of \texttt{session\_ok} message from \texttt{sender}
1. if \texttt{session\_ok} has been received from all existing children then
2. set current state to ready
3. send \texttt{session\_ok} to \texttt{parent}
4. end if

Upon retransmit timer timeout
1. if there are more than \texttt{threshold} timeouts then
2. remove all children who have not sent \texttt{session\_ok} back
3. set current state to ready
4. send \texttt{session\_ok} to \texttt{parent}
5. else if all children have sent \texttt{session\_ok} back then
6. send \texttt{session\_ok} to \texttt{parent}
7. set current state to ready
8. else
9. for each child that has not sent \texttt{session\_ok} back do; send \texttt{session} to the child; end for
10. restart retransmit timer
11. end if

Figure 5.7: General tree construction algorithm.
5.3.2 The randk Algorithm

The randk algorithm propagates the session message to random nodes selected from the membership view. This algorithm is very simple, and it only relies on the gossip-based membership protocol we discussed in Section 5.2. However, because of the randomness in children selection, the coverage of the algorithm is probabilistic. In fact, assume that the membership view of a node represents a uniform random sampling of the whole system, and the “fanout” $k$ of the algorithm is $\Omega(c + \log N)$, the probability that the algorithm can have complete coverage is $e^{-e^{-c}}$. In the above, $c$ is a constant and $N$ is the system size. For example, for a system with $N = 2000$ nodes, if each node propagates $k = 8$ tree construction messages, the probability that an on-demand overlay covers all nodes is just above 0.5, even when the system has no failures.

5.3.3 The leafset+RF Algorithm

The randk algorithm can only achieve high coverage with large fanout $k$, even when the system has no failures. This is because nodes only propagate the construction message randomly. If we look at a specific node, then the probability that it receives at least one tree construction message is $1 - (1 - 1/N)^{k-N} \rightarrow 1 - e^{-k}$.

To achieve high coverage with small $k$ (which means small construction overhead), we have designed a new algorithm called leafset+RF (here RF means random forwarding). The idea is to combine random forwarding with some kind of deterministic forwarding for the tree construction message. Specifically, this algorithm requires some augmentation to the membership layer. In addition to the random membership view, each node also maintains a “leafset”. The leafset of a node $p$ consists of the $l$ nodes whose logical IDs are closest to that of $p$ (half with larger IDs and half with smaller IDs). Note in a dynamic system, a node may not always have the “correct” leafset. For example, in Figure 5.8, node $H$ has failed, but nodes $G$ and $I$ still have it in their membership view. Also, node $B$ should have node
C as its right neighbor (i.e., C should be in B’s leafset). However, due to the dynamics in the system, B may not even be aware of C (and nodes D, E and F). As a result, B may consider G as its right neighbor. As B gossips with other nodes, B may learn about new nodes whose IDs are closer than its leafset neighbors. In that case it can replace its current leafset neighbors with the newly discovered nodes. When there are no failures, it is likely that nodes will quickly find their true leafset neighbors.

With the random membership view and the leafset neighbors, the leafset+RF algorithm builds on-demand overlay trees as follows. The general steps to build on-demand trees are the same as in Figure 5.7. However, when each node selects children nodes, it will select \( k_0 \) nodes from its leafset and \( k - k' \) from its random membership view. The corresponding message propagation is called “leafset forwarding” and “random forwarding”, respectively. Assume \( l = 2 \), i.e., each node keeps one left neighbor and one right neighbor. If the system has no failures, the left/right pointers will connect nodes into a global ring. At this time, the tree construction will cover all nodes as long as \( k' \geq 1 \). In a dynamic system, it is unlikely the whole system will be connected into a ring. For example, the leafset entries of a node may point to a node that has failed, or nodes that are not its “true” leafset neighbors. However, if we consider the directed graph \( G = (V, E) \) formed by the live nodes and their \( k' \) closest leafset pointers (i.e., \( V \) is the set of live nodes, and \( E = \{(u,v) | v \) is a leafset neighbor of \( u; u,v \in V\} \)), the graph will be partitioned into a set of “segments”, where each segment is a strongly connected subgraph of \( G \). If each node propagates the tree construction message to its \( k' \)
leafset neighbors as well as to some random nodes, a segment will be completely covered, if and only if at least one node in the segment receives a random propagation message (from nodes outside the segment). Therefore, the coverage of the leafset+RF algorithm depends on the likelihood that each segment receives at least one random propagation message.

Consider a segment $S$ with $s$ nodes. Assume that the random forwarding part of the algorithm can cover at least $\frac{N}{2}$ nodes. Each of these nodes will forward $k - k'$ random messages. The probability that at least one of these is received by a node in segment $S$ is

$$1 - \left(1 - \frac{s}{N}\right)^{(k-k')\cdot\frac{N}{2}} \to 1 - e^{-(k-k')s/2} \quad (5.6)$$

Suppose $k - k' = 4$ (i.e., each node propagates the tree construction message to 4 random nodes), if $s = 1$, the probability that the segment gets covered is $1 - e^{-4/2} = 0.865$. However, if $s = 10$, the probability that it gets covered is $1 - e^{-40/2} = 1 - 2 \cdot 10^{-9}$, which is very close to 1. Thus we can see that the probability of a segment gets covered depends on its size. The larger the segment size, the more likely it will be covered. If we increase $k'$, it is likely the graph $G$ will consist of fewer, but larger segments. As a result, increasing $k'$ may improve the coverage. However, if the total fanout $k$ is fixed, increasing $k'$ may reduce the random forwarding degree $k - k'$. This may be undesirable since first, it will reduce the probability that a segment receives at least one random forwarding message (as indicated in Equation(5.6)). Second, random forwarding can reduce the height of the on-demand tree. Forwarding entirely along leafset neighbors would result in very tall trees.

To improve the locality awareness of the overlay being constructed, when a node $p$ forwards the tree construction message randomly, it can favor nodes that have a small network delay from itself. Since the node IDs are randomly selected, the nearby nodes of $p$ are likely to have very different IDs, thus likely to be in different segments. This variant of the algorithm is called leafset+RF+LOC.

In the above discussion, we assume the node IDs are randomly assigned. Random node
ID assignment means the average delay between a node and its leafset neighbors are likely to be large. For service applications where many nodes could be clustered together (e.g., a service deployed in a small number of data centers), it may be possible to assign similar IDs to nodes in the same cluster. As a result, nodes close to each other in the ID space are also close to each other in the network. This may improve the performance of the on-demand overlays, although from failure resilience point of view, this may be undesirable since few nodes will propagate messages outside their clusters.

Our leafset concept is similar to that of the Pastry DHT [70]. However, the leafset of a Pastry node must be accurate in order to route messages correctly, while the leafset of our membership layer can contain inaccurate information. Such inaccuracy is dealt with using redundant message at overlay construction time.

5.3.4 The tree+RF Algorithm

The leafset+RF algorithm relies on both deterministic and random forwarding to ensure complete coverage with high probability. In this section, we explore another algorithm called tree+RF that also provides both deterministic and random forwarding. For this algorithm, the membership layer maintains a tree structure in addition to the random membership view. Each node will have a parent and multiple children neighbors. The tree can be maintained using any existing protocol (e.g., the one used by ESM [38]). As we have argued, maintaining a tree overlay for a long time is difficult. In fact, that is our motivation to build on-demand overlay networks. However, notice here the tree structure is only used for membership management. It is not directly used for the application level processing. As a result, there is no requirement that the tree be correctly maintained all the time. For example, the tree may be temporarily partitioned or have loops. All we want is that the tree edges will connect nodes into multiple segments, similar to the leafset+RF algorithm.

The tree+RF algorithm builds on-demand overlay following the steps as the general algorithm. However, whenever a node propagates the tree construction message, it will
propagate the message to all its tree neighbors in the membership layer, and propagate the rest messages to random nodes (each node propagates $k$ messages in total).

Although the membership tree is likely to connect nodes into multiple segments, one difference from the membership ring (as used by the leafset+RF algorithm) is that since most nodes in a tree are close to the leaf nodes, when there are node failures, it is likely to produce small segments. For example, in Figure 5.9, if node $B$ fails, the tree will be partitioned into four segments, with three of them consisting of single node (i.e., nodes $E$, $F$ and $G$). These single node segments are very difficult to cover using random forwarding, as is the case for the rand$k$ algorithm.

5.3.5 On-demand DAG construction

We have only discussed on-demand tree construction. It is known that tree overlays are vulnerable to network and node failures. Any interior node or link failure would cause the tree overlay to partition. As a result, we may want to build DAG (direct acyclic graph) overlays for the query and control command execution due to the redundant links in the overlays. The tree construction algorithms can in fact be adapted to build DAG overlays. Specifically, each node still propagates the overlay construction message as before. Whenever a node receives an overlay construction message, if it currently has enough parents, it will respond with a prune message. Otherwise it will accept the sender of the message as an additional parent. Because each node may have multiple parents, the failure of small number of nodes and overlay links are unlikely to cause any live nodes to be disconnected.
5.4 Application Management with MON

In this section, we present the management capabilities of MON. We describe (1) the SQL-like language we have implemented for distributed status query and control; (2) a client-side API that can integrate MON queries with higher level programming logical; and (3) how to use these capabilities for application management.

5.4.1 MON Query Language

Assume an on-demand tree has been constructed, a query or control command can be executed as follows. First, starting from the initiator (tree root), each node will propagate the command to its children nodes. Second, each node will execute the query locally. Third, when a node receives the result from all of its children, it will aggregate the results with its local result, and send the aggregate data to its parent. For example, if the query wants to compute the average CPU load of all nodes, each node will compute a tuple \((n, \text{total load})\), where \(n\) means the number of nodes in its subtree, and the \(\text{total load}\) is the sum of CPU load. When the root node gets such a tuple, it can easily compute the average CPU load for its subtree.

For DAG overlays, each node will have multiple parents. However, when a node \(p\) first receives a query from a parent \(q\), it will choose \(q\) as the primary parent and send the aggregate result to \(q\). For other parents, \(p\) will send an empty result message to them. This way, each node is only counted once in the final aggregation result.

We note that the local execution of the query is very generic, it can query the operating system, the file system (file content as well as file metadata), the application process, or even a networked server. For example, in our MON deployment on the PlanetLab [62], the local execution of a query often needs to query the CoMon [5] server via the HTTP protocol, and then extract the desired metric from the returned HTML webpage.

We have implemented a SQL-like language that allows application developers to query
and control the status of the distributed application. The SQL-like language presents a
database view of the distributed application to the application developer. The whole dis-
tributed system is regarded as a database table, each application node is a row in this table
and each status metric is a column. Such database view allows the application developer to
focus on expressing what data they want to query, rather than how the query is executed
(in a distributed fashion).

The general language syntax for aggregate queries looks like the following:

\[
\text{select } \text{agg}(<\text{resource}>) \ [\text{where } <\text{condition}>]
\]

Here \textit{agg} is the aggregation function. We currently support three kinds of aggregation
functions: \texttt{AVG}, \texttt{TOP-K} and \texttt{HISTOGRAM}. \textit{resource} is the metrics that we want to query. It can
be simple metrics such as CPU load and free memory. It can also be complex metrics that
have parameters. For example, \texttt{filesize(“mon.log“)} queries the size of the file with the name
“mon.log”, and \texttt{procmem(“mon“)} and \texttt{proccpu(“mon“)} query the memory and CPU usage of
the process “mon”. The metric can also be some internal metric of the application process,
if it provides some query interface (e.g., we have instrumented the FreePastry [6] system and
used MON to query its internal status such as the average “proximities” of different routing
table rows, and the number of live routing table entries). \textit{condition} is a boolean expression
over different resources (we have implemented only the conjunctive normal form (CNF)
boolean expressions. However, it is known any boolean expression can be transformed to
the CNF form). A command is locally executed on a node only if the \textit{condition} evaluates
to true. For example

\[
\text{select } \text{avg}($\text{freemem}$) \text{ where } \text{load} > 10
\]

will compute the average free memory for the nodes with a CPU load greater than 10.

Non-aggregate queries generally look like the following:

\[
\text{select } <\text{resource\_list}> \text{ where } <\text{condition}>
\]
Here resource_list is a list of one or more resources. The command should return the specified resource values on the nodes that satisfy the condition. Note the where clause is mandatory for non-aggregate queries. This is meant to remind the user that non-aggregate queries may return too much data. Therefore, the user should provide a where clause to limit the amount of data returned.

The third category of commands is for status control. Right now we have provided the capability to execute any shell command on all the nodes. The general syntax is like the following:

\[
\text{select run(cmd) [where \langle condition\rangle]}
\]

It means the shell command cmd should be executed on any nodes that satisfy the condition. To facilitate the execution of common shell commands, we also implemented some higher level commands such as

\[
\text{select grep(keyword, file) [where \langle condition\rangle]}
\]

It will try to search the specified keyword in the specified file, and return the first line of match \(^2\).

### 5.4.2 MON API and Scripts

We have a command line client that allows users to interactively query and control their applications. In addition, we have provided a client side C++ API so that MON can be integrated into higher level programming languages for automated application management. The API consists of two simple function calls:

1. \text{mon.init();}
   
   this initiates the appropriate data structures.

2. \text{mon.exec(char*cmd, MonResult* result);}

\(^2\)Other operators, such as returning the first K lines of match, the last K lines of match, or random K lines of match can also be implemented.
this executes a command (in the syntax described before) and waits for the results. The API can be easily integrated with some extensible scripting language such as Python, so that users can write high level scripts. For instance, the following script periodically queries the average CPU load on a set of nodes, and take some additional actions if the average load is greater than some threshold.

```python
while(1) {
    create_session();
    avg_load = mon_exec("select avg(load)")
    if(avg_load > 10){
        hosts = mon_exec("select top 10 load");
        //do something else
    }
    stop_session();
    //sleep some time
}
```

We can see that compared with traditional management tools, MON provides a mechanism to query and control a large distributed application in a holistic fashion. Such powerful capability is likely to help application developers quickly detect and debug any potential problems in their applications.

### 5.4.3 Querying the Application Internal Status

For the purpose of detecting and diagnosing potential problems that a distributed application has encountered, it is extremely helpful if one can query the fine grained internal status of the application processes while they are running. The basic mechanisms of MON can be used for querying application internal status. In fact, only the local execution needs to be extended to interact with the application process. The query propagation and result aggregation can remain unchanged.
In order to make its internal status available, the application needs to provide some query interface. This can be in the form of shared memory or some networked interface. For example, the application may listen on a socket and accepts queries from the socket. Once a query is received, the application can compute the desired metric and return it to MON. MON does not even need to understand the semantics of the metric. Instrumenting the application to provide such query interface involves only minimal efforts. In addition, if the application was developed using our reusable PPF framework (see Chapter 4), the framework already provides a management interface. All the application needs to do is to implement a callback function that computes the requested metric value.

5.4.4 Distributed Log Query

For legacy applications, it may not be possible to instrument the application code. However, many applications will produce a lot of useful information in their log files. As a result, MON can be used to query the distributed log files instantly. For example, one can query all the log files to search for the keyword “Error”, this might allow the application developer to find out if any application node has encountered some failure conditions.

5.5 Evaluation

In this section, we present simulation and PlanetLab results to evaluate the tree construction algorithms of MON. For simulation, the total number of nodes is \( N = 2000 \), each time we let the system evolve until steady state is reached. We then randomly kill certain percentage of the nodes and build on-demand overlay trees. For PlanetLab experiments, we deploy MON on about 320 PlanetLab nodes and create on-demand trees using different algorithms. We then evaluate the coverage and response time of the on-demand trees. We also compare the reliability of tree overlays and DAG overlays.

Figure 5.10(a) shows the average node coverage for the algorithms. We can see when
there are no failures, leafset+RF, leafset+RF+LOC and tree+RF can achieve 100% coverage. However, the randk algorithm will miss some of the nodes, even when the system has no failures.

Figure 5.10(b) shows the probability of full coverage for the same algorithm. We can see that when the percentage of node failures increases, the probability for tree+RF quickly decreases. The reason is that node failures may introduce small segments in the membership tree, which are difficult to be covered by random forwarding.

Figure 5.10(c) shows the response time for the algorithms. We can see that the leafset+RF+LOC
has the smallest response time for small percentage of node failures. When the failure rate is high, \texttt{leafset+RF} and \texttt{leafset+RF+LOC} may create trees with large height (due to the message propagation along the membership ring), as a result, the response time may be higher.

Figure 5.11 shows the experiment results on the PlanetLab. For this experiment, we deploy MON on about 324 nodes on the PlanetLab, and build on-demand trees using the \texttt{randk} algorithm with fanout \( k = 4, 5, 6, 8 \) and the \texttt{Leafset+RF+LOC} algorithm with the fanout \( k = 4 \). Figure 5.11(a) shows that the \texttt{Leafset+RF+LOC} algorithm can cover more than 318 nodes on average, while the \texttt{randk} algorithm can only about 305 nodes for \( k = 4 \) (i.e., the same message overhead). Even when \( k = 8 \), the coverage of the \texttt{randk} algorithm is still lower than that of the \texttt{Leafset+RF+LOC} algorithm.

Note the \texttt{Leafset+RF+LOC} algorithm may use slightly more bandwidth than \texttt{randk} for membership maintenance, due to the maintenance of a leafset in addition to random membership view. However, this additional bandwidth overhead is very small. In our implementation, each node exchanges heartbeat messages with its leafset neighbors every 10 seconds, and each message is much less than 100 bytes. Thus the additional overhead is on the order of tens of bytes per second. We believe such small extra bandwidth during the normal time is worthwhile in return for the better coverage and lower message overhead at overlay construction time.

Figure 5.11(b) shows the average response time for the same experiment. We can see that the \texttt{Leafset+RF+LOC} algorithm not only covers more nodes, but has better average response time. On average the response time is about 18\% lower than the \texttt{randk} algorithm with \( k = 4 \).

Since MON builds on-demand overlays and do not repair failures, a question is how long these on-demand overlays can be used without major failures. We define the \textit{session reliability} of an on-demand overlay as the probability that the overlay is still \textit{usable} after \( t \) time units. An overlay is \textit{usable}, if at most \textit{max\_drop} nodes have been disconnected.
from the overlay since it was initially constructed. $\max_{\text{drop}}$ is a parameter that the application developer can specify before the overlay is constructed. We have conducted experiments on the PlanetLab to study the session reliability of on-demand overlay networks. For this experiment, we build on-demand trees and DAGs (with different maximum number of parents, or “fanin”). Figure 5.12 shows the session reliability for tree overlays and DAG overlays with different maximum number of parents (with $\max_{\text{drop}} = 5$). As the figure shows, the session reliability of tree overlays are very poor. In a realistic environment such as the PlanetLab, the probability that a tree overlay is usable for 20 minutes is less than 50%, while the probability for a DAG to be still usable is more than 80%, even though each

\begin{footnote}{Nodes can be disconnected from the overlay, if they have failed or if their parents have failed.}
node can have at most 3 parents.

We have also conducted experiments using MON to query distributed log files. For this experiment, we created a log file on each of about 320 machines. The file sizes ranged from about 400KB to about 5MB, with an average of just over 2MB. Only six of these log files contain the word “Fail”. These six machines had the following delay (rtt) to our local node: 0.3ms, 19ms, 27ms, 35ms, 59ms, 62ms. We call these nodes the “interesting nodes”. We used MON to execute the `grep` command to find out on which nodes the keyword “Fail” appeared in the log file. The query was executed about 1500 times.

Figure 5.13 shows the CDF of the number of interesting nodes discovered. We can see about 10% of the time we discovered all 6 interesting nodes, about 40% of the time we discovered at least 5, and about 89% of the time we discovered at least 4. The average number discovered is 4.36. The average execution time of these log queries was just over 2 seconds. MON does not return all interesting nodes every time. This is because we use UDP for the communication between nodes, and message loss can cause the result from some nodes to be missing from the final result. For performance reasons, we have used a relatively small retransmission count of 6. This retransmission count can be specified when executing the query command, thus it allows the application developer to trade off between response time and the completeness of the query result.
Chapter 6

InfoEye: Self-Configuring Information Management

The previous chapter presented MON, a system for dynamic distributed status query and control. Dynamic query is necessary for distributed system/application management, because it allows the application developer to query (pull) any information available in the system. However, if there are some metrics that need to be queried repeatedly, continuous monitoring (push) may be more efficient in bandwidth usage since it does not need to propagate the query every time. This chapter presents InfoEye, a system that can dynamically recognize application query patterns such as query arrival rate and attribute popularity, and automatically configure itself to use either push or pull for different metrics in order to minimize information management overhead. InfoEye differs from MON in that it targets multi-attribute range queries instead of aggregate queries, and it focuses on algorithms for dynamical configuration rather than decentralized communication and collaboration.

6.1 InfoEye Overview

InfoEye considers a distributed system with $N$ nodes. Each node has a large number of dynamic attributes that may change over time. To manage such distributed system, the system managers or applications running on the system may want to query its dynamic information. Thus an information management system is needed that can track the distributed system information and answer information queries efficiently.

Figure 6.1 shows a typical distributed system consisting of (1) system nodes that execute different application tasks; (2) management nodes that monitor the status of all sys-
tem nodes and perform management tasks (e.g., job scheduling, resource allocation, system trouble-shooting); (3) monitoring sensors that monitor and provide the information of each system node to management nodes. The information management system resides within the management nodes, which can resolve information queries from other system management modules or user applications.

To resolve information queries, the information management system can obtain the dynamic information using either push or pull. Push means the monitoring sensors periodically report their local information to the management nodes. As a result, when a query is received, the management node already has the necessary information to resolve the query. Pull means no information is proactively reported to the system nodes. Instead, when a query is received, the management node can dynamically query the monitoring sensors to obtain the necessary information. It is clear that the effectiveness of push and pull depends on the application query patterns such as query arrival rate and attribute popularity. Therefore, an ideal information management system should be able to dynamically configure itself based on the statistical query patterns and systems conditions in order to minimize the monitoring overhead.

In the rest of this chapter, we first present a more formal description of our target system, we then formulate the problem of dynamic configuration and develop analytical models and algorithms to solve the problem. Finally, we present the evaluation results
6.2 The InfoEye Model

In this section, we present the InfoEye query model, the statistical query patterns and problem formulations. The notations used in this chapter are summarized in Table 6.1.

### 6.2.1 InfoEye Queries

We consider a distributed system as shown in Figure 6.1. In a real system there can be multiple management nodes. In this chapter, we focus on exploiting statistical application patterns and consider the algorithm in a single management node. To extend the idea to multiple management nodes, the management nodes may need to share statistical information with each other, depending on how they partition workload among themselves. We leave this as our future work.

For applications such as resource discovery and management, the query can often be expressed as locating some system nodes that have certain resources, e.g., \((a_1 \in [l_1, h_1]) \land (a_2 \in [l_2, h_2]) \land \cdots (a_k \in [l_k, h_k])\), where \(l_i\) and \(h_i\) are the desired lower bound and upper bound for \(a_i\), respectively. Each query can also specify the number of system nodes that are needed. The query answer should return the specified number of system nodes, each of which is based on simulation and prototype implementation.

<table>
<thead>
<tr>
<th>notation</th>
<th>meaning</th>
<th>notation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>total number of system nodes</td>
<td>(a_i)</td>
<td>system state attribute</td>
</tr>
<tr>
<td>(A)</td>
<td>set of all attributes</td>
<td>(A^*)</td>
<td>subset of attributes to be pushed</td>
</tr>
<tr>
<td>(f_1 = \frac{</td>
<td>A^*</td>
<td>}{</td>
<td>A</td>
</tr>
<tr>
<td>(T_i^*)</td>
<td>optimal push interval for (a_i)</td>
<td>(T_i)</td>
<td>staleness constraint of a query</td>
</tr>
<tr>
<td>(S_1)</td>
<td>size of push message</td>
<td>(S_2)</td>
<td>size of probe message</td>
</tr>
<tr>
<td>(\lambda)</td>
<td>average query arrival rate</td>
<td>(n)</td>
<td>average probing overhead</td>
</tr>
<tr>
<td>(p_1)</td>
<td>% of resolvable queries using (A^*)</td>
<td>(l_i)</td>
<td>lower bound requirement for (a_i)</td>
</tr>
<tr>
<td>(l_i^*)</td>
<td>(optimal) push threshold for (a_i)</td>
<td>(f_2)</td>
<td>% nodes in the push subspace</td>
</tr>
<tr>
<td>(p_2)</td>
<td>% queries in the push subspace</td>
<td>(p_3)</td>
<td>% queries satisfied by the push intervals</td>
</tr>
</tbody>
</table>

Table 6.1: Notations.
satisfies the query predicate. Finally, each query can also specify a staleness constraint $T_i$ on a required attribute $a_i$, which means the attribute value used to resolve this query should be no more than $T_i$ seconds old. The staleness constraint is meant to give applications more specific control on their query result. If a query does not specify such constraint, a default value (e.g., 30 seconds) can be used instead.

On each system node, there is a monitoring software called a monitoring sensor. The monitoring sensor can be configured by the management node to periodically push its information only when certain conditions are satisfied \(^1\). It can also respond to a dynamic probe with its current information. Such configurability allows the management node to achieve adaptiveness based on statistical query patterns.

6.2.2 Statistical Patterns

InfoEye performs automatic self-configuration based on dynamically maintained statistical information about the queries and system conditions. Specifically, the current InfoEye system maintains the following statistical information:

**Frequently queried attributes.** Although system nodes can be associated with many attributes, it is likely only a subset of them are frequently queried by current applications. For example, in distributed applications where computing jobs are mainly CPU-bound, most queries will specify requirements on the CPU resource, but not on other attributes. This means the management node can configure the monitoring sensors to only push the subset of attributes (denoted as $A^*$) that are likely to be queried. This allows the management node to resolve queries that only specify attributes in $A^*$. For other queries, dynamic probe (pull) can be invoked for their resolution.

**Frequently queried range values.** Besides selecting popular attributes, we can further reduce the system cost by filtering out unqualified attribute values. For example, if most

\(^1\)This means each monitoring sensor is very simple. It only waits for configuration from the management node, without making sophisticated monitoring decisions.
queries on CPU time require a node to have at least 20% free CPU time, the nodes with less than 10% free CPU time do not need to push their CPU value since they are unlikely to satisfy the query predicate. Generally, we can configure the monitoring sensor with a push triggering range $[l_i, \infty)$ for each selected attribute $a_i \in A^*$. The monitoring sensor will push the attribute data only if the attribute value falls into this range. The lower bound $l_i$ of the configured range is called the *push threshold* for the attribute. By properly setting the push threshold, we can filter out a lot of unnecessary data push without significantly decreasing the query hit ratio (i.e., percentage of queries that can be resolved by the pushed data).

Figure 6.2 illustrates the problem of push threshold selection for one attribute. The solid line is the cumulative distribution function (CDF) of an attribute $a_1$ across all $N$ nodes, and the dashed line is the CDF of the lower bound requirements from the current queries. As the figure shows, 90% queries require the attribute to be greater than $l$, and only 74% of nodes satisfy this requirement. If we configure the push threshold to be $l$, 74% of nodes will push their attribute data and 90% of queries can be resolved by the pushed data. However, if we increase the push threshold from $l$ to $l'$, only 20% of nodes need to push their attribute data with a moderate decrease of query hit ratio (from 90% to 65%).

**Frequent staleness constraints.** The last query pattern that InfoEye utilizes is called frequent staleness constraints. When an application makes a query, it can specify a *staleness constraint* $T_i$, which means the attribute data used to resolve the query should be no more

---

2 The query predicates such as in resource queries often do not have upper-bound constraints. Our scheme can also be easily extended to include a finite upper-bound.
than $T_i$ seconds old for attribute $a_i$. It is likely for any attribute $a_i \in A^*$, different queries may have different staleness requirements. As a result, the push interval (i.e., update period) of $a_i$ should be dynamically configured, so that the push frequency is just enough to satisfy the staleness constraints of most queries.

Node attribute distributions. In addition to the query patterns, InfoEye also maintains an estimate of node attribute distribution. The distribution can be used for two purposes. First, we can estimate the probing cost (i.e., the number of probes that will be generated) based on the node attribute distributions. Second, the attribute distributions allow us to estimate the push cost reduction and pull cost increase when we configure the push thresholds for different attributes (in Section 6.3.2). Since our system involves multiple attributes, we maintain multi-dimensional histograms to estimate the attribute distribution, which can be obtained by executing infrequent aggregate queries (e.g., histogram) over all the nodes [49].

6.2.3 Problem Formulations

Since InfoEye combines the push and pull for data collection, its management cost (or total system cost) includes two parts, push cost and pull cost. The push cost is the amount of data periodically delivered from different system nodes to the management node. The pull cost is the amount of data generated per time unit for pulling the attribute data, in response to queries that cannot be resolved by the management node locally. The goal of InfoEye is to dynamically configure the monitoring sensors, so that the total system cost is minimized.

Corresponding to the application query patterns, there are three configuration parameters that InfoEye can tune. The first is the subset $A^*$ of attributes that are pushed. This means

---

3Notice here we are adopting an application driven approach. Even though different attributes may change their values at different speed (e.g., CPU changes much faster than disk), we do not explicitly look at the attributes. Instead, we assume application queries can specify the right freshness requirements, and we will configure the push interval based on the query requirement. This is desirable since InfoEye does not need to estimate the change rate of the attribute values and the impact of different change rate on the applications.
each monitoring sensor only periodically pushes a subset $A^*$ of attributes. When a query arrives, if all attributes it specifies are in $A^*$, no additional cost is incurred. Otherwise, some on-demand probing protocol is needed to find enough nodes that satisfy the query \(4\).

Since each monitoring sensor periodically (every $T$ seconds) pushes $f_1 = \frac{|A^*|}{|A|}$ percent of the attributes, assume the message size is proportional to the number of attributes pushed, and $S_1$ is the size of the message if all $|A|$ attributes are pushed, the push cost of the system can be expressed as $\frac{1}{T} N f_1 S_1$ (bytes/second). Suppose the average query arrival rate is $\lambda$ and on average we need to probe $n$ nodes with $2n$ messages (probes and replies) to resolve a query by pull. Let $p_1$ denote the query hit ratio, and $S_2$ denote the size of a probe message \(5\), the pull cost of the whole system is $2n(1 - p_1)\lambda S_2$. As a result, if only popular attributes are configured, and $A^*$ is the set of selected attributes, the total system cost is

$$\frac{1}{T} N f_1 S_1 + 2n(1 - p_1)\lambda S_2. \quad (6.1)$$

Given a subset $A^*$ that has been selected, we can further reduce the system cost by selecting a push threshold $l^*_i$ for each attribute $a_i \in A^*$, and filtering out the nodes that do not satisfy the push thresholds. The set of push thresholds define a subspace $\{(a_1, a_2, \cdots, a_{|A^*|})|a_i \geq l^*_i, 1 \leq i \leq |A^*|\}$ in the $|A^*|$-dimensional space. We say a node is “covered” by the subspace, if its value for each attribute $a_i \in A^*$ is above the push threshold. We say a query is “covered” by the subspace, if its lower bound requirement on each $a_i \in A^*$ is above the push threshold. If a query is covered by the subspace, it means all the nodes that satisfy the query (called the answer set of the query) are covered by the subspace, thus it can be locally resolved safely. For a query not covered by the subspace, its answer set is not completely resolved safely.

\(4\)There are different ways for dynamic probing, e.g., using random sampling or on-demand spanning trees such as MON. Regardless the particular probing protocol, we assume in order to resolve a query by probing, on average $n$ nodes need to be contacted with $2n$ messages. In practice, $n$ can be obtained from previous probes.

\(5\)Since it is unlikely for a query to specify requirements on many attributes [16], we assume the message size for both probe and reply is $S_2$, which is a constant much smaller than $S_1$. However, this is only for notational simplicity and is not essential to our model.
available. In this case, we assume a probe is invoked, so that the query result is not biased toward a subset of the answer set.

Suppose a system node reports its attribute data $A^*$ only if the node is covered by the subspace, and $f_2$ percent of the system nodes are covered by the subspace defined by the push thresholds. The push cost of the system is reduced to $\frac{1}{T} f_2 N f_1 S_1$ since only the $f_2$ percent of the nodes do periodic push. Correspondingly, if $p_2$ percent of the queries (among those that only specify attributes in $A^*$) are covered by the subspace, a total of $(1 - p_1 p_2)$ percent queries need to be resolved by dynamic pull. As a result, the total system cost becomes

$$\frac{1}{T} f_2 N f_1 S_1 + 2n(1 - p_2 p_1)\lambda S_2.$$  \hspace{1cm} (6.2)

To further reduce the system cost, each system node can push the value of $a_i \in A^*$ every $T_i$ seconds when the value is above the push threshold. The push cost for attribute $a_i$ becomes $\frac{1}{T_i} f_2 N \frac{S_i}{|A|}$. Thus, the total push cost for all selected attributes is $\sum_{a_i \in A^*} \frac{1}{T_i} f_2 N \frac{S_i}{|A|}$. Suppose under the above configuration, $p_3$ percent of queries (out of the $p_2 p_1$ percent of queries that specify attributes in $A^*$ and are covered by the subspace defined by the push thresholds) can satisfy their staleness constraints. Then a total of $(1 - p_3 p_2 p_1)$ percent queries need to invoke pull operations. Thus, finally the total system cost for all three configuration parameters is

$$\sum_{a_i \in A^*} \left( \frac{1}{T_i} f_2 N \frac{S_i}{|A|} \right) + 2n(1 - p_3 p_2 p_1)\lambda S_2.$$  \hspace{1cm} (6.3)

### 6.3 Design and Algorithms

In this section, we describe our algorithms to achieve optimal information monitoring based on the formulas derived in the previous section. Our goal is to minimize the total system cost in Equation (6.3). For simplicity, we describe the algorithm in several steps as follows. In practice, the steps are executed in an iterative fashion.
6.3.1 Push Attribute Selection

The goal of push attribute selection is to select a subset of attributes \( A^* \subseteq A \), so that the total system cost is minimized. According to Equation (6.1), \( A^* \) can affect the push cost (i.e., \( f_1 = |A^*|/|A| \) percent of complete attribute push cost) and the percentage \( p_1 \) of queries that can be resolved by the management node locally. Larger \( A^* \) implies a larger push cost but also a larger query hit ratio, while smaller \( A^* \) implies smaller push cost but also lower query hit ratio and thus higher pull cost.\(^6\)

Our push attribute selection algorithm is shown in Figure 6.3. In the figure, \( C \) is the collection of attribute subsets, each corresponding to a set of queries (e.g., \( A_1 = \{a_1, a_3\} \) corresponds to all queries that specify requirement on \( a_1 \) and \( a_3 \)). \( freq(A_i) \) is called the “query frequency” for \( A_i \), which means the percentage of all queries that are represented by \( A_i \). \( freq'(A_i) = \sum_{A_j \subseteq A_i} freq(A_j) \) is called the “cumulative query frequency”, which means

\(^6\)Note in our configuration algorithms, we only consider minimizing the total bandwidth cost. Since queries that are resolved by pull can incur larger response time (i.e., time since the query was issued until the query results are received), we can specify additional constraints on how much queries must be resolved locally. This only slightly changes our problem formulation. As a result, our algorithms can still be used to solve the problems with minor modifications.
the percentage of queries that can be resolved by the push data if all attributes in \( A_i \) are pushed. Given an \( A_i \), if we push all attributes in \( A_i \), we will increase the push cost by \( \frac{1}{T} N |A_i| S_1 \), but we also reduce the pull cost by \( 2n \cdot freq'(A_i) \lambda S_2 \), because \( freq'(A_i) \) percent of queries can now be locally resolved. The decrease in pull cost minus increase in push cost is called the “cost reduction”, which indicates how the system cost will change if \( A_i \) is pushed.

Initially, we set \( A^* \) to be empty, which means no attribute is pushed. Thereafter, we repeatedly select the subset \( A_i \) with the largest cost reduction, and add \( A_i \) to \( A^* \). This is repeated until either all attributes have been added to \( A^* \), or the cost reduction for any remaining attribute subset is negative. Note when \( A_i \) is added to \( A^* \), its attributes should be removed from all other subsets in \( C \). This may create duplicate subsets in \( C \). For example, after the attributes in \( A_i = \{a_1, a_2\} \) are removed, the two remaining subsets \( \{a_1, a_3\} \) and \( \{a_2, a_3\} \) will be the same as each other. These subsets are then merged, and the cumulative query frequency recomputed.

### 6.3.2 Push Threshold Configuration

Given the subset \( A^* \) as selected by the push attribute selection algorithm, the push threshold configuration algorithm should select a push threshold \( l_i^* \) for each attribute \( a_i \in A^* \), so that the total cost as in Equation (6.2) is minimized.

The idea behind push threshold configuration is similar to push attribute selection. For each attribute \( a_i \in A^* \), we normalize the possible value range to \([0, 1.0]\), and divide the range into steps of size \( d \). Initially all the push thresholds are set to 0, which means every node will push its attributes in \( A^* \). At each step, an attribute \( a_i \) is selected, and its push threshold increased from \( l_i^* \) to \( l_i^* + d \). Such an increase will reduce the push cost since fewer system nodes are covered by the subspace. However, it also increases the pull cost since more queries are uncovered by the subspace. Thus, at each step the attribute \( a_i \) is selected in a way that maximizes the net cost reduction. The above process is repeated until either
every push threshold has reached its maximum, or there is no attribute with positive cost reduction.

The pseudo code for the push threshold selection algorithm is shown in Figure 6.4. The main difference from push threshold selection is how to compute cost reduction given a particular configuration. In the algorithm, $B$ and $B'$ are the histogram bins for the node attribute and query distribution. Each bin in $B$ or $B'$ is described by a tuple of $|A^*| + 1$ fields. The first $|A^*|$ fields define the bin, and the last field is the percentage of nodes/queries in the bin. For example, $b = (v_1, v_2, \ldots, v_{|A^*|}, 0.1) \in B$ means 10% of the machines have attribute $a_i \in [v_i, v_i + d), 1 \leq i \leq |A^*|$.

### 6.3.3 Push Interval Selection

Given the selected push attributes $A^*$ and push thresholds $\{l_i^* | a_i \in A^*\}$, The goal of push interval selection is to select push interval $T_i^*$ for each attribute $a_i \in A^*$, so that the total system cost according to Equation (6.3) is minimized.

The push interval selection algorithm works the same way as the previous two algorithms and thus it is only briefly described here. Initially, the push interval $T_i^*$ for each $a_i \in A^*$ is set to a minimum value (i.e., this is the smallest interval that monitoring sensors can push attribute data periodically). Thereafter, at each step, an attribute is selected and
the corresponding push interval incremented (by some constant step size). The attribute is selected so that the increase of its push interval results in the largest cost reduction. This is repeated until every push interval has reached some maximum value, or the increase of any push interval would result in negative cost reduction. The cost reduction is computed as the reduced push cost due to slower push minus the increased pull cost due to more queries being pulled (because their staleness constraint cannot be satisfied by the pushed data).

6.3.4 Practical Issues

There are several practical issues that need to be mentioned about our algorithms. First, when we resolve a query by pull, the pulled data can be cached for future query resolution. However, this is unlikely to have a big impact on our query resolution, since the data are not periodically refreshed, thus will timeout within a short period of time. Second, the push interval selection assumes each attribute is independently pushed. This may be undesirable due to a lot of small messages. This can be solved as follows. Suppose the set of push intervals have been selected, and the smallest push interval is $T_i^*$, we can normalize every $T_j^*$ to $T_i^* \left\lfloor \frac{T_j^*}{T_i^*} \right\rfloor$, which is the largest multiple of $T_i^*$ that is still $\leq T_j^*$. This way other attributes can be piggybacked to the push messages for $a_i$.

6.4 Experimental Evaluation

In this section we present an experimental evaluation of InfoEye system. We first describe our simulation methodology and results, then present the prototype implementation of InfoEye and our experiment results from the PlanetLab [62].

6.4.1 Evaluation Methodology

Our simulator consists of a query generator that can generate a range of different kinds of query workload, a query collection that captures the statistical query patterns, and three
configuration modules (i.e., popular attribute selection, push threshold configuration, and push interval configuration). Unless otherwise specified, the system size is \( N = 3000 \), the default push interval is \( T = 30 \) seconds, the total number of attributes is \( |A| = 50 \), the number of nodes to be probed for each pull is \( n = 50 \), the push packet size is \( S_1 = 1000 \) bytes and the probe packet size is \( S_2 = 100 \) bytes. Our parameters are chosen to represent a “typical” system. For example, in the CoMon [5] monitoring service currently running on the PlanetLab, each resource report contains more than 40 attributes, and has about 900 bytes.\(^7\)

Our query generator uses similar methods as previous work [58] for query generation. For each query, we first decide the number of attributes in a query, which is uniformly distributed between \([1, k], 1 \leq k \leq |A|\). Next, the specified number of attributes are selected from \( A \). The probability that an attribute is selected follows the Zipf [18] distribution. After that, the lower bound on each attribute is generated. We assume that the value range of each attribute is divided into 50 equal sized bins (intervals). The lower bound for an attribute is generated according to a Zipf distribution, but biased toward the highest value. To generate

\(^7\)CoMon is essentially a push-based system. In order to minimize the monitoring overhead, the push interval is set to 5 minutes.
node attribute values, we use a probability distribution that mimics the actual attribute
distribution we observed on the PlanetLab, namely, most nodes have moderate attribute
values, but a small number of nodes will have very large or very small attribute values.

We use the total system cost defined in Section 6.2.3 as the main evaluation metric. For
each experiment, we first generate a set of “training queries” (usually 2000 of them) using
the query generator. The query arrival follows a Poisson process with a mean arrival rate $\lambda$.
We then run our algorithms to configure the InfoEye system (i.e., to select push attributes,
push thresholds, and push intervals). Next, we generate another set of “validation queries”
according to the same model, and resolve the queries against our system configuration. The
cost of the system for resolving the validation queries is computed. Each experiment is
repeated 200 times, and the average cost is reported.

We mainly compare the system cost of InfoEye to that of the two static approaches, pure
push and pure pull. In pure push-based systems, each monitoring sensor periodically reports
all attribute data using the default push interval. Thus, the system cost is independent of
the query arrivals. In pure pull-based systems, no periodic information push is involved,
thus the system cost is proportional to the rate of query arrivals.

6.4.2 Simulation Results

Figure 6.5 shows the performance of our attribute selection algorithm. Figure 6.5(a) shows
the system cost of InfoEye for different query arrival rate and maximum number of attributes
$k$ in a query. Figure 6.5(b) shows the number of attributes selected for push. The results
show that InfoEye consistently performs better than both pure push and pull approaches.
When the query arrival rate is small, pure push involves a lot of unnecessary overhead. At
this time, InfoEye can configure the monitoring sensors to push only a small number of most
popular attributes (as shown in Figure 6.5(b)), and achieve a small system cost similar to
pure pull. When the query arrival rate increases, the cost of pure pull increases linearly.
However, InfoEye can configure the monitoring sensors to push more attributes. As a result,
its system cost is always smaller than either pure push or pure pull. If the system is statically configured, the system cost would be many times that of InfoEye for either small or large query arrival rates.

Figure 6.6 shows system cost when both attribute selection and push threshold selection are applied. The node attribute data are generated using the distribution described in Section 6.4.1, and the “moderate value” $v$ is 5. We can see that when the query arrival rate $\lambda$ is small, the cost of InfoEye is similar to Figure 6.5(a). This is because when $\lambda$ is small, $|A^*|$ is small. As a result, the system cost is dominated by pulling attributes that are not in $A^*$. However, when $\lambda$ is large, more attributes are pushed, and the effect of push threshold selection becomes more significant. Figure 6.7 shows the system cost when
all three algorithms are applied. The push interval for pure push is $T = 30$ seconds. The query staleness requirement follows a distribution similar to that used for node attribute distribution. The minimum requirement is 30 seconds, the maximum requirement is 180 seconds, and the moderate value is 50 seconds. Figure 6.7 shows that by pushing the attributes at a frequency that satisfies most (but not all) query requirement, we can further reduce the system cost so that even when the query arrival rate is large, the total cost of InfoEye is about 25% smaller than pure push \(^8\).

We now examine the adaptivity of InfoEye, i.e., its ability to re-configure itself in response to dynamic query pattern changes. We only show the results of push attribute reconfiguration.

\(^8\)Figure 6.7 shows that when $\lambda$ is large, the cost for $k = 5$ is can actually be smaller than $k = 3$. This is because for $k = 5$, more attributes are pushed as indicated in Figure 6.5. As a result, push interval selection has more space for improving the push cost.
due to the space limitation. Figure 6.8 shows the adaptivity of InfoEye when the query arrival rate changes. For this experiment, initially the mean query arrival rate is 8 queries/second. After the initial configuration, we generate validation queries and record the total system cost every 10 seconds. An exponential weighted moving average of this “instant cost” is then compared with the system cost predicted by Equation (6.1). If the difference between the two costs exceeds 20%, a re-configuration is initiated. For this experiment, we also use a “historical query window” of the recent 2000 queries. System re-configuration is based on these historical queries. At time 400, we change the query arrival from 8 to 12. Figure 6.8 shows that the higher query arrival rate results in higher system cost. At time 470, InfoEye detects the system change and re-configures itself to push more attributes. Although push cost is increased, the total system cost is reduced since less queries need to be resolved by pull. Figure 6.8 also shows the cost of pure push and pull. We can see when the query arrival rate is 8, the cost of InfoEye is close to that of pure pull. Both are much smaller than pure push. After the reconfiguration, the cost of InfoEye is close to that of pure push, and both are much smaller than that of pure pull. Figure 6.9 shows the adaptivity of InfoEye to attribute popularity changes in the queries. The experiment settings are similar to the previous one, except the mean query arrival rate is 10 for the whole experiment. At time 400, we switch the popularity of the top three and bottom three attributes. We observe that InfoEye can quickly detect this change and reconfigure itself. Because at this time, the history queries are a mixture of two different patterns, only a smaller number of attributes are selected. After another 120 seconds or so, most queries in the history window are from the new distribution. As a result, InfoEye reconfigures again and selects the right subset of $A^*$ for push.

6.4.3 Prototype Results

We have implemented a prototype of our InfoEye system and deployed it on the Planet-Lab [62] testbed. We have a monitoring sensor on each PlanetLab node, which can peri-
odically check the local resource attributes and push the data to the management node. The management node is responsible for storing the pushed attribute data and answering queries. Currently the data are stored in some simple memory resident data structures. For really large distributed systems, the management node may use some database servers to store the dynamic data. The management node is responsible for running the configuration algorithms and configure the monitoring sensors based on the computed system parameters such as the push threshold for each attribute. Currently we have only integrated the push threshold selection algorithm with our management node. In addition to the monitoring sensors and the management node, we have a query client. This query client again generates synthetic queries and send the queries to the management node. The management node and query client are run on a local machine.

Our experiments involve about 280 PlanetLab nodes. Each monitoring sensor samples the local resource values every 10 seconds, and compares them with the configured push thresholds. If the resource values are greater, the attribute data are pushed to the management node. The management node accepts the pushed data and answers queries. It also invokes the push threshold selection algorithm every 60 seconds \(^9\). The new push thresholds are then sent to all monitoring sensors. The query client can generate queries of different patterns and send the queries to the management node. Each query specifies requirements on three attributes: available CPU time, amount of free memory, and amount of free disk space. The management node keeps a sliding window of past 1000 queries for the push threshold configuration. Each time before the configuration, the management node also runs a global aggregation query to get the node attribute distribution for the whole system. Under the above settings (e.g., 280 nodes and 1000 historical queries), each configuration run takes about 3ms, and the memory consumption of the management node is under 5MB.

For the first experiment, we first let the query client generate queries that require small

\(^9\)System reconfiguration can be triggered by either a timer or any changes in system parameters. Our current prototype only implements the timer-triggered reconfiguration.
amount of CPU time, free memory and disk space. Specifically, the lower bound for these attributes are randomly distributed within \([10\%, 20\%]\), \([10\text{MB}, 20\text{MB}]\) and \([10\text{GB}, 20\text{GB}]\), respectively. After about 12 minutes, the query pattern is changed. The queries now require a minimum of CPU, free memory and disk space that are randomly distributed within \([20\%, 30\%]\), \([20\text{MB}, 30\text{MB}]\) and \([20\text{GB}, 30\text{GB}]\), respectively. The query arrival rate is 4 per second for the entire experiment. Figure 6.10 shows the push threshold configured by the management node every minute. Initially the push threshold for CPU time is configured to be a little less than 10\%. After the pattern change, the push threshold is configured to be a little less than 20\%. The push threshold for free memory and disk space show similar trend and are therefore omitted. From Figure 6.11 we can see the effect of such system configuration. Initially, since the push threshold is low, about 80\% of the nodes need to
periodically push their attributes. When the query pattern has changed and the queries require more resources, less nodes can satisfy the queries. Our push threshold selection algorithm correctly recognizes this, and configures the push thresholds to higher values. This results in only about 30% of the nodes periodically push their attribute data. Although this means a small proportion of queries \((1 - p_2)\) have to be resolved by pull, the overall system cost is reduced, due to large savings in the push cost.

Figure 6.12 and Figure 6.13 show the same results for a different query pattern change. For this experiment, during the first 15 minutes, the queries are generated just like the first experiment. Thereafter, the query distribution is not changed, but the mean query arrival rate is changed to 2. Figure 6.12 shows when the query arrival rate decreases, the configured push threshold for CPU is increased. The reason is that a smaller query arrival rate means
a smaller overhead for query pull. As a result, the system cost can be reduced by slightly increasing the push threshold, which leads to smaller percentage of nodes that periodically push their data, and a smaller percentage of queries that need to invoke pull operations (as shown in Figure 6.13).

6.5 Discussions

We have presented the design and evaluation of InfoEye, a novel model based, self-adaptive distributed information management system. The goal of InfoEye is to resolve multi-attribute queries in large-scale dynamic distributed systems with minimum monitoring overhead. To achieve this goal, InfoEye maintains statistical information about both application queries and node attribute distributions, and dynamically configures itself to achieve minimum management overhead. Through extensive simulation studies, we show that InfoEye can achieve much lower management overhead than static solutions. More importantly, when the query pattern changes, InfoEye can quickly re-configure itself to adapt to the changes. We have also implemented a prototype of the InfoEye system and validated the feasibility and performance on a real network environment. A web based query interface is also provided at http://cairo.cs.uiuc.edu/projects/monitoring/.
Chapter 7

Related Work

In this chapter, we discuss research work generally related to the design, implementation and management of large distributed applications, and highlight the difference of our work from previous research.

7.1 Structured and Unstructured Overlay Networks

Large distributed applications are often designed around the concept of overlay networks. The overlay networks built by early file sharing applications such as Gnutella [13] and KaZaA [7] are often characterized as unstructured overlay networks. Such overlays are easy to maintain, since there are no invariants that must be maintained. In unstructured overlay networks, it is relatively easy and efficient to locate popular objects, which are replicated by many peers. However, for rare objects, there is no guarantee that such objects can be found without flooding the whole network. Structured overlay networks, or distributed hashtables (DHTs) [74, 65, 70, 80, 35, 55, 53] are initially motivated by such drawbacks of unstructured overlays. In most DHTs, nodes are organized into certain regular graph topologies, and files are assigned to nodes using mechanisms such as consistent hashing. Many DHTs allow the lookup of a given file in $O(\log N)$ time when each node maintains $O(\log N)$ neighbors. But some DHTs [65, 35] can trade off between the amount of state maintained by each node and the lookup performance.

There has been work [27] that attempts to unify the APIs provided by different DHTs. The goal is to use DHTs as a common “routing substrate” for different applications. Al-
though this routing layer facilitates the communication between different application nodes, it also means the application does not have direct control on how the application level data are transported. As a result, the performance and QoS requirements of the application may be violated. This is why none of the existing media streaming applications [39, 79, 10], which have strict QoS requirements, have been built on top of DHTs.

The OpenDHT [43, 68] project has attempted to provide DHT as a service. The goal is to maintain one DHT and share it among multiple applications. It facilitates the development of distributed applications because the application developer no longer needs to maintain a DHT. However, since OpenDHT provides data plane service, again it may impose performance bottlenecks on distributed applications. Also, since the service interface has to be generic, therefore, low-level get and put interfaces are provided. In comparison, our RandPeer is a control plane service. It facilitates neighbor selection of distributed applications, without imposing restrictions on how the application nodes manage their data. Also, since RandPeer is a specialized service, it can provide higher level service interfaces such as register and lookup.

Overall, we believe DHTs are valuable for their self-organization and scalability. However, directly building applications on top of DHTs may not be desirable, due to the various QoS and performance requirements of the applications. However, building control plane services on top of DHTs might be attractive, because the performance requirements on control plane services are usually less stringent, and more meaningful service interfaces can be provided.

7.2 Gossip Protocols

Gossip protocols [28] are initially designed for replicated database maintenance. By making use of randomization and message redundancy, gossip protocols guarantee that with high probability, a gossip message will eventually be received by all nodes in the system, despite message loss and individual node crashes. Due to their simplicity and probabilistic guar-
Gossip protocols are a class of randomized algorithms. Randomization is an effective approach to dealing with unexpected failures in a wide area environment. For example, the overlay construction of the MON system uses both deterministic propagation and randomized propagation to ensure high coverage despite imperfect membership information, and peers registering with RandPeer will choose random node IDs to ensure that the membership trie is roughly balanced.

7.3 Implementing Large Distributed Applications

Implementing a large distributed application that can run in a wide area environment involves a lot of engineering challenges. As a result, much work has attempted to simplify application implementation. MACEDON [69] (and its followup project MACE [46]) allow application developers to implement their distributed algorithms in a domain specific language, the implementation is then translated into C++ code for compilation. MACEDON and our PPF framework are similar in that they both use the single thread, event driven architecture for supporting both simulation and real world execution. However, PPF allows application developers to program in C++ itself, rather than some high level, domain specific language. This makes the debugging of the code straightforward, since the code is not translated. In contrast, using MACEDON, application developers may have to read translated code or inspect the high level implementation.

P2 [52] is a system that proposes to build overlay networks by declarative programming
language. Application developers implement their algorithms using the Overlog (similar to Datalog) language. The “implementation” is then translated into running code. Compared with MACEDON/MACE and PPF, P2 presents even higher learning curve, since most system developers are unfamiliar with logic programming languages. Further, due to the semantic gap between the specification and automatically generated code, it is even more difficult to debug tricky distributed application problems (e.g. timing issues) and fine tune the code for performance optimization.

FreePastry [70] is similar to PPF in that it also provides programming support for both simulation and real world mode. The difference is that FreePastry provides DHT APIs, while PPF provides APIs similar to the socket API, we believe a socket-like API is more likely to be accepted due to the flexibility that they provided.

Jones et. al [42] have elaborated on many of the challenging issues involved in implementing real world large distributed applications. They have also discussed some design principles such as using single code base, supporting both simulation and real world execution, and providing message level simulation. However, they have only provided general discussion, while PPF is a concrete framework that can be downloaded and re-used for implementing new applications.

7.4 P2P Media Streaming

P2P media streaming has attracted much attention recently, due to its ability to utilize the residue bandwidth of peers and its potential to scale to large number of receivers. However, due to the highly dynamic nature of a P2P environment, designing a practical P2P streaming system that has good scalability, failure resilience and QoS/performance has been a difficult task. Tree based or multi-tree based systems [39, 15, 75, 59, 22] are vulnerable to node failures. Mesh-based systems [79, 67, 60] are more resilient to failures, but cannot guarantee the overall network connectivity, especially when network locality is desired. We have presented
the design of DagStream, which builds DAG structures for P2P media streaming. Similar to mesh-based systems, DAG structures allows each peer to stream data from multiple other nodes (parents). Different from these systems, however, the global network connectivity can be guaranteed, as long as each peer maintains a given number of parents. Thus peers can improve their locality awareness, without worrying about network partitionings.

7.5 Information Monitoring and Distributed Application Management

The emergence of large distributed applications has made the management of such applications an important problem. Traditional management systems [25, 5, 54] have focused on managing the computing infrastructure itself. As a result, they all focus on continuously monitoring a predefined set of status metrics. For example, the CoMon [5] system provides monitoring service on the PlanetLab [62] wide area testbed. CoMon continuously monitors several tens of metrics on the PlanetLab nodes, and has been very useful to PlanetLab users. However, currently the metrics that CoMon monitors are mostly system level metrics, and already the monitoring interval is set to five minutes to reduce monitoring overhead. If CoMon were to continuously monitor more detailed information about applications running on the PlanetLab, the overhead would be excessively high. This problem can be solved by dynamic query, because the dynamic information is only obtained when they are needed.

Research on information query in large scale systems has attracted much attention recently. Early file sharing systems provided simple keyword based search in unstructured overlay networks. In an attempt to address inefficiency of unstructured overlay networks, many DHT systems have been proposed. While DHTs have much better scalability and efficiency, the query is limited to exact match. As a result, much research has attempted to support various complex queries such as range queries [32, 81, 58], similarity queries [33], and more general complex queries [36]. For example, Gao et. al. [33] have proposed to embed a
distributed kd tree on the DHT for similarity search (or nearest neighbor search), and the PIER [40] system has attempted to build general query engine on a distributed hashtable. Our goal in the MON system is to support application management. As a result, we support aggregate queries rather than general database queries (e.g., joins between different tables). This allows application developers to quickly detect any potential problems in a large distributed system. We also adopted the novel on-demand approach for overlay construction and query execution, this is in contrast to most existing systems built on top of DHTs.

MON targets aggregate information query. Astrolabe [77] and SDIMS [78] are two systems that support scalable information aggregation for large distributed systems. However, Astrolabe uses gossip protocols to propagate status information, which means it may take a long time for the query result to converge to the true value. SDIMS is built on a DHT, thus it not only has larger overhead, but also has the drawbacks of persistent overlays. Some systems [41] use gossip protocol itself for aggregation. The aggregation protocol is very simple. However, the queries are limited to simple queries without any conditions.

MON focuses on dynamic query. It is possible for a large distributed system, some metrics are frequently queried while many other metrics are rarely queried. Our InfoEye system attempts to combine dynamic query (pull) and continuous monitoring (push) in order to minimize the information management overhead.

Combining push and pull-based information access has been explored by some previous work in different contexts such as delivering dynamic web objects to clients [29] and collecting data in a sensor network [76]. Although the general idea of combining push and pull is not new, we should emphasize applying the idea to a specific environment requires non-trivial system analysis and design. In our case, it means identifying application query patterns and deriving the analytical model for total system cost, which makes it possible for adaptive push/pull configuration.
Chapter 8

Concluding Remarks

8.1 Conclusion

Large distributed applications based on peer-to-peer technology have the potential to achieve high scalability, availability, reliability and QoS/performance for Internet scale services. However, the design, implementation and management of such applications is a challenging process, due to the lack of design principles, software architectures, reusable code bases and powerful management tools. In an attempt to simplify such process, this dissertation has made the following contributions.

- We have proposed a layered overlay construction and maintenance architecture (OCMA) for designing large distributed applications. OCMA decomposes a complex application into smaller components (layers). Such a decomposition simplifies the application design and facilitates the reuse of different components, yet still provides application developers maximum flexibility as to how each layer should be designed. We have demonstrated the utility of OCMA by building two large distributed applications, the DagStream system for locality aware P2P streaming and the Management Overlay Networks (MON) system for distributed management. Both conform to the OCMA layered architecture, yet both have explored novel designs such as on-demand overlay construction and control plane services.

- We have provided PPF, a reusable framework for implementing large distributed applications. PPF simplifies application implementation because it frees the application
developer from the details of asynchronous network programming, yet the same code can run in both simulation and real world mode. The latter is especially useful for the initial debugging of an application and for the transitioning from simulation to real world deployment.

- We have implemented the Management Overlay Networks (MON), a tool for dynamically query and control the status of a large distributed application running in a wide area environment. Such distributed status query and control allows the application developers to quickly detect potential application problems and take control actions. MON is currently deployed on the PlanetLab [62] and offers public service to the PlanetLab community. We also expand the dynamic query capability of MON and build the InfoEye system that can combine dynamic query and continuous monitoring to minimize information management overhead.

8.2 Future Work

Although we have made contributions to all development phases of large distributed applications, there still exist many challenges.

- **Designing secure large distributed applications** The current OCMA architecture has not taken security into account, yet security is an essential part of large distributed applications as they are used to provide more and more critical Internet services. It is likely that security needs to be built in at each layer of the OCMA architecture. However, the question is which security mechanisms should be implemented at which layer, considering the end-to-end security requirement and the tradeoff between security and performance.

- **Novel control plane services** The use of control plane services is a promising approach to simplifying application design and improving application performance/security.
Our RandPeer system has provided control plane membership management. The Chubby lock service from Google has provided synchronization service. The question is what other control plane operations can be implemented as separate services, and how these services can be implemented. Since control plane services are often small scale, it is likely novel techniques can be used for their design and implementation.

- **Distributed event logging and replay** Our PPF framework supports local simulation and debugging. However, it is possible that an application may encounter unexpected failures when it is running in a real world environment. In order to debug the application under unexpected failures, it is important to add sophisticated logging facility to the PPF framework, so that a running application will log its input and output events (e.g., network messages), and replay such traces in a purely simulation environment, where the execution can be paused and repeated for debugging purpose.

- **PlanetOS** Our MON system has provided the ability to dynamically query and control the status of large distributed applications. However, the management of large distributed applications may require more system support than just query and control. As a result, an integrated environment with easy to use tools for deploying, monitoring and controlling distributed applications is needed. We call such an environment PlanetOS, meaning that it should make the use of a planet scale distributed system as simple as the use of a single computer.
References


[58] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In WORLDS’04, December 2004.


[71] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In 18th ACM Symposium on Operating Systems Principles (SOSP 2001), October 2001.


Author’s Biography

Jin Liang was born and raised in Sichuan Province, China. He received his Bachelor of Engineering degree in Computer Science from the University of Science and Technology of China (USTC) in 1998, and his Master of Engineering degree from the Institute of Software, Chinese Academy of Sciences in 2001. Since August 2001, he has been a Ph.D. student in Computer Science at the University of Illinois at Urbana-Champaign. He joined Professor Klara Nahrstedt’s MONET research group in 2002. His research interests are in large scale distributed computing systems such as peer-to-peer media streaming and distributed service management. In his research, he has designed novel techniques such as control plane services and on-demand overlay networks for large distributed systems.