Scalable Cluster Technologies for Mission-Critical Enterprise Computing

Werner H.P. Vogels

VRIJE UNIVERSITEIT

Scalable Cluster Technologies for Mission-Critical Enterprise Computing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 11 november 2003 om 13.45 uur
in het aula van de universiteit,
De Boelelaan 1105

door

Werner Hans Peter Vogels

geboren te Ermelo

promotoren: prof.dr.ir. H.E. Bal
prof.dr. A.S. Tanenbaum

In rememberance of those
who would have loved to
read about this work

Rob & Trees

Cor van den Biggelaar

Alje van der Laan

Wilhelmus Vogels

ISBN: 1-4116-0166-1

Chapter 2 has been published in the *Proceedings of the 15th ACM Symposium on Operating Systems Principles,* Copper Mountain, CO, December 1995.

Chapter 3 has been published in *IEEE Computer,* Volume 31 Number 11, November 1998.

Chapter 4 has been published in the *Proceedings of the 8th IEEE Hot Interconnets Symposium*, Stanford, CA, August 2000.

Chapter 5 has been published in *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

Chapter 6 has been published in *Proceedings of the 2nd International Enterprise Distributed Object Computing Conference*, San Diego, November, 1998.

Chapter 7 has been published in *Proceedings of the Second Usenix Windows NT Symposium, Seattle*, WA, August 1998.

Chapter 8 has been published in the *Proceedings of the IEEE International Conference on Cluster Computing: Cluster-2000*, Chemnitz, Germany, December 2000.

Chapter 9 has been published in the *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

On the cover is the SGI Origin 3000 Lomax cluster at NASA Ames research center. It is a 512-processor, single system image, distributed shared memory system used for performing scalability studies, benchmarking, and solving large-scale problems for the Information Power Grid. The system has 256 Gigabytes of main memory, 2 Terabytes of disk storage and can reach a peak of 409 GFLOPS.

The cluster consists of 128 C-Bricks, each with quad 400 MHz R12000 processors and 2Gb of local memory, and 44 R-Bricks for communication. Additional details on the Lomax cluster can be found at *http://www.nas.nasa.gov/User/Systemsdocs/O3K/hardware.html*

The photo was made by Tom Trower of NASA Ames Research Center

Table of Contents

Acknowledgements

Annette, Laura and Kim have contributed more to the work described in this thesis than anyone will ever realize. They deserve credit for stimulating me to look for solutions in creative, non-traditional ways, for motivating me to face new challenges, and for always encouraging me to take all the time I need. The supreme happiness in life is the conviction that we are loved.

First and foremost my gratitude goes to Robbert van Renesse, Ken Birman and Fred Schneider for being colleagues, mentors, students, critics and friends during the past 10 years.

If you are fortunate you will meet people on your path that have the ability to point you into the right direction and as such have a profound influence on the work without getting much recognition. Jim Gray, on several occasions, has steered me towards succeeding, even where progress was not obvious. His unwavering support has been much appreciated. He is a good friend.

Paulo Verissimo and Luis Rodrigues took me into their research team at INESC, when I still had little to show for, and allowed me to develop my skills, publish my first works and build an international network of colleagues and friends. I am grateful for their support in those days and for the opportunity to experience Portugal.

My first true collaboration at Cornell was with Thorsten von Eicken. In a rather intensive year we architected and built U-Net, which had an exciting mix of creative innovation and superb engineering. I have very fond memories of that year and the interaction has been the blueprint for many collaborations that followed afterwards.

It are not only mentors and teachers that have influenced my work in the past years, my students have played an equal enlightening role. I got more than I deserved when my star students Dan Dumitiru, Pino Decandia, Ivano Rosero and Mark Rogge decided to join me at Reliable Network Solutions, Inc. More recently it has been Chris Re who constantly challenges me to think laterally.

I was able to develop my interest in computer science research because I was given opportunities to do so during my years at the Haagse Hoogeschool. Rob Kraaijenbrink and Kees van Loon supported my drive for discovery, and allowed me to keep the PDP-11 alive until PCs could no longer be ignored. I hope that my friend and colleague from those days, Henk Schouten, will also be able to fulfill his dreams.

I had a life before computer science, but Frank de Leeuw with his persistent advice and Annette coming back from London with an 8 bit Atari ended that life.

I am grateful to Annette for her help with editing and reviewing the thesis.

A special word of thanks goes to Henri Bal and Andy Tanenbaum for their efforts to make this dissertation become a reality. They had the difficult task to review work they had not been closely involved with and did so with an open mind and an eye for detail. Thanks to their effort, the structure and content of the thesis improved significantly.

Finally I would like to thank my reading committee; Ken Birman, Jim Gray, Thilo Kielman, Keith Marzullo and Willy Zwaenepoel, for their detailed reading of the thesis and attaching their seal of approval.

Support

The research presented in this thesis has been supported by:

- The US Defense Advanced Research Projects Agency's Information Technology Office under contracts ONR-N00014-92-J-1866, ONR- N0014-96-1-10014, AFRL-F30602-99-1-0532 and AFRL-F49620-02-1-0233.
- The National Science Foundation under Grant No. EIA 97-03470.
- The National Aeronautics & Space Administration under the Remote Exploration and Experimentation program administered by the Jet Propulsion Laboratory.
- The Microsoft Corporation through grants by the Scalable Servers group of the Bay Area Research Center and by the Microsoft Server product division.
- The Intel Corporation though grants by the Server Architecture Lab.

Chapter 1

General Introduction

Enterprise computing has changed significantly in the past decade. In the past, the workloads at corporate datacenters were dominated by centralized processing using a limited number of big database servers, mainly handling online transaction processing (OLTP) and batch processing tasks in support of the back-office process. The organization of these datacenters has evolved from mainframe-oriented into a large collection of flexible application servers providing a very diverse set of services. These services still include the traditional order processing and inventory control, but now also provide internal and external information portals, continuous data-mining operations, pervasive integration of customer relationship management information, email and other collaboration services, and a variety of computational services such as financial forecasting.

An important observation is that these services have become essential to the successful operation of the enterprise, and that any service interruption, either through failure or through performance degradation, could bring the activities of the enterprise to a halt. The mission-critical nature of these services requires them to be scalable, highly-available, and with robust performance guarantees. Organizing these services into compute clusters appeared a logical step as cluster technology held the promise of cost-effective scalability and was considered to be a good basis for implementing a highly-available service organization.

In the early 1990's the technologies provided by traditional cluster computing, being either OLTP or parallel computing oriented, were insufficient for developing the scalable, robust services the new information-centered enterprise required. The problems that faced enterprise cluster computing are best described by Greg Pfister in the conclusion section of his book “*In search of Clusters – the ongoing battle in lowly parallel computing*” [77]:

> *Attempts to use overtly parallel processing have previously been crippled by wimpy microprocessors, slothful communication, and the need to rebuild painfully complex parallel software from scratch.*
>
> *The result of this situation has been the completely justifiable conviction that this form of computing simply was not worth the trouble unless it provided enormous gains in performance or function. With a performance focus came a fixation on massive parallelism; with a functional focus came a fixation on massively distributed processing.*

In this thesis some of the results of my research into the problems that faced mission-critical cluster computing are presented. The problems ranged from enabling off-the-shelf workstations and operating systems to exploit high-performance interconnects, to structuring the management of geographically distributed clusters. These problems inhibited the wide-spread adoption of cluster computing as a solution for scalable and robust enterprise computing. The research in this thesis has mainly focused on the following four areas:

- Unlocking the full potential of high-performance networking
- Developing efficient runtime systems for cluster-aware enterprise applications
- Structuring the management of large-scale enterprise computing systems
- System analysis through large-scale usage monitoring

The new technologies that resulted from my research are considered to have been major contributions to the ability to build scalable clusters in support of modern mission-critical enterprise computing. These technologies have transitioned into industry standards such as the Virtual Interface Architecture, into commercial available clustered application-servers, and into the design of a new commercial highly-scalable cluster management system that supports enterprise-wide, geographically distributed, management of cluster services.

This thesis presents several of the more important results, but the research has lead to many more results, also outside of the four main areas, and these results are referenced in Appendix A.

Each of the four main research areas is described in more detail in the following sections.

1.1 Unlocking the full potential of high-performance networking

Although significant progress had been made in the early 1990's in developing high-performance cluster interconnects, this technology was not yet suitable for integration into off-the-shelf enterprise cluster systems. The communication technology was targeted towards high-performance parallel computing where the operating systems used styles of application structuring that made it difficult to transfer the technology to standard workstations. For example the IBM SP2 used techniques that allowed only one application access to the interconnect, as the operating system provided no protection on the network.

The arrival of standard high-performance network technology brought the promise that regular workstations and servers could use high-performance communication in a manner similar to parallel computing systems, but in a much more cost-effective way. Unfortunately the standard operating systems were not structured to support high-performance and the overhead on network processing was so high that most of the benefits of the new networks could not be made available.

In 1994 I started research, in collaboration with Thorsten von Eicken, to break through this barrier. The resulting architecture, *U-Net*, presented a new abstraction that combined the power of direct user-level network access with the full protection of standard operating systems. In U-Net the network adapter was virtualized into the application's address space, enabling end-to-end network performance close to the bare-wire maximum. The complete separation of data and control for network processing enabled the construction of very high performance cluster applications.

An industry consortium lead by Intel, Microsoft and Compaq standardized the U-Net architecture into the Virtual Interface Architecture, which became the de-facto standard for enterprise cluster interconnections. The U-Net architecture, its transition into VIA, and experiences with large production clusters based on VIA can be found in Part I of this thesis.

1.2 Building efficient runtime systems for enterprise cluster applications

Advances in the scalability of cluster hardware and cluster management systems enabled a large set of applications to benefit from improved performance and availability. Many of these applications were based on legacy systems which in themselves were not "aware" of the cluster environment on which they were executing. Even though this transparency appeared to have several advantages, especial when applied to client-server interaction, it was impossible to meet the demands of scalability and fault-tolerance for server applications without introducing at least some awareness about the distributed nature of the system it was executing on. These insights were similar to our experiences with building group structuring systems for complex distributed production systems, where the conclusion was reached that server-side transparency blocked the development of advanced distributed applications.

My research into efficient runtime systems for cluster-aware applications focused on what would be the right tools for application developers to structure their applications if distribution was made explicit instead of transparent. This research resulted in the development of an application-server (dubbed *Quintet*) intended to serve the application-tier of multi-tier enterprise applications. Quintet provides tools for the distribution and replication of server components to meet availability, performance and scalability guarantees. The approach in Quintet is radically different from previous systems that support object replication, in that the replication and distribution are no longer transparent and are brought under full control of the developer.

A number of the components that were developed for Quintet found their way into commercial application-servers, and Quintet was the basis for the application development support in the Galaxy Cluster Management Framework. Other tools such as the multi-level failure detector have been reused in the farm and cluster management system, but also transitioned into stand-alone tools for tracking small web and compute-farms.

Part II of this thesis is dedicated to the research on cluster runtime and management systems. In chapter 5 our experiences with the use of academic software in production systems are described, while in chapter 6 an overview is given of the design of the Quintet application-server.

1.3 Structuring the management of large-scale enterprise computing systems

The improvements in processor and network performance in the early 1990's were by themselves not sufficient to bring cluster computing into the mainstream. Software support for cluster computing, both at the systems level and at the application level, faced serious structuring and scalability problems. This was most visible in the area of cluster management where the enterprise support systems were targeted towards small-scale clusters and supported only very specific hardware configurations.

Whereas low-end parallel computing for the masses was being enabled by the *Beowulf* cluster management tools, there was no similar solution for enterprise cluster computing. The main reason for the lack of progress was that software systems for enterprise cluster management had to serve a variety of application types, all with very specific demands for the way cluster management was performed. The distributed systems components necessary to build these management systems are often more complex than the applications they have to serve.

The *Galaxy Cluster Management Framework* was the first system to provide a scalable solution for the management of clusters in large data-centers. Its multi-tier management infrastructure enabled the management of compute-farms at multiple locations, with within each farm islands of specialized clusters that are managed according to a cluster profile. The system was designed based on principles that were the result of an analysis of the production use of our communication software and an in-depth analysis of existing management systems. Galaxy was successful in that its design and principles were selected by a major operating systems vendor as the foundation for its next generation cluster system. The work leading to Galaxy and a detailed description of the system can be found chapters 7 and 8 in part II of this thesis.

1.4 System analysis through large-scale usage monitoring

Understanding the way systems are used in production settings is essential for systems research that seeks to find solutions for the practical problems facing the computing industry. Monitoring large-scale systems and the analysis of the results is almost a research area by itself. The design of a monitoring and analysis system is often very complex as there are many concurrent data sources involved, and there is only limited or no control over the system usage. A difficult challenge when designing the instrumentation and monitoring of a system is to ensure that it produces *complete* usage information such that rigorous statistical analysis is possible.

This thesis contains two studies performed by me that use large scale experimentation and monitoring:

The first is a study into the behavior under overload conditions of a large-scale cluster interconnect, which consisted of 40 individual switches. This study uses a traditional experiment design, where there is full control over behavior of the data sources and sinks. The challenge with the design of these experiments was that only observations at the sinks could be used to draw conclusions about the behavior of the switches in the middle of the network, given that the switches themselves could not be instrumented. What complicated the experimentation was the large amount of data that needed to be collected at high speeds to be able to perform proper analysis later in the process. The details of this study are in chapter 4.

In the second study, to which part III of the thesis is dedicated, I was confronted with the problems of continuously monitoring a production environment where there is no control over the data sources. The subject of this study was the usage characteristics of file-systems of standard workstations. To this purpose the operating systems of a large number of workstations were instrumented and observed over a longer period of time. Again in this study the analysis was complicated by the magnitude of the amount of data collected and the complexity introduced by the heterogeneity of the applications execution on the workstations. In the conclusions of this study I stress the importance of specialized statistical techniques for processing large sets of observation data. This exact, rigorous statistical analysis, which was missing from earlier file-system studies, is essential for the construction of proper workload models, both for future benchmarking and for advanced systems tuning.

Part I

An Architecture for Protected User-Level Communication

Introduction

In the early 1990's it was becoming clear that the acceptance of cluster computing as a key component in the evolution of enterprise computing was stagnating. The main reason for the stagnation was that advances in network technology are not translated into an improved networking experience for cluster applications. The advances in network performance were deemed essential for clusters to be constructed out of regular workstations, an approach which would make scaling clusters through a rack & stack approach cost-effective. The only modular architectures which provided some form of improved performance were those targeted at the scientific computing market, such as the Thinking Machines CM-5 and the Meiko CS-2. These architectures were not suitable for the enterprise computing market as they exhibited the traditional problem seen at many parallel computing platforms of using CPUs that are at least one generation behind the CPUs available for the general workstation market. This problem combined with the often exorbitant costs of these clusters made it very difficult to find a place in the enterprise market for cluster computing. The only cluster architecture with some limited enterprise computing successes was the IBM SP-2, but the proprietary nature of the communication infrastructure, as well as the lack of support for protected access to the network interface reduced the wide-spread adoption of the architecture outside of the scientific computing arena.

The advances in workstation networking were initially based on a surge of popularity of ATM networking, which was becoming commonly available for both SBUS and PCI architectures. The expectation was that end-to-end latency would be in the 20-100 µsec range and bandwidth would be 100 Mbit/sec or more. In reality the minimum latency measured was in the order of hundreds of microseconds, and full bandwidth could only be achieved under infinite source conditions using large messages. When using simple standard protocols such as UDP as transport over the high-performance network, the latency for smaller messages was similar to or worse than the latency achieved over 10 Mbit/sec Ethernet [37].

A new architecture for high-performance networking

The main reason for the lack of performance improvement was the message processing overhead incurred during the send and receive actions. The operating system structures for network processing had basically not changed since the late 1970's when most network communication went over slow serial lines. The new low latency networks exposed the fixed overhead that was associated with each receive, which was almost independent of the number of bytes transferred. The properties of the new high-performance networks could not be exploited without a complete overhaul of the way communication primitives were offered in traditional operating systems.

In 1994 I started a research project together with Thorsten von Eicken to re-architect the way standard operating systems handled high-performance interconnects. The resulting *U-Net* architecture addressed the issues by delivering data directly to the applications, bypassing the operating system completely during the send and receive operations. At the same time U-Net guaranteed that multiple processes could use the network concurrently in a fully protected fashion. The prototype system provided end-to-end performance that was close to the performance of the network itself. The U-Net architecture was first presented at the 1995 ACM Symposium on Operating Systems Principles (SOSP-15) and is described in detail in chapter 2.

A foundation for the Virtual Interface Architecture

U-Net did not only address the issues of operating systems performance, but also offered a new approach to the way that the networked applications could be structured, by completely separating the data and control paths. It went further than architectures such as Active Messages and Fast Messages in removing the last software coating that these architectures had put over the interaction with the network. In U-Net the network interface was virtualized and each application was exposed to a set of queue structures that resembled the way networking hardware normally operates.

The U-Net architecture was seen by the enterprise cluster architects as the major step forward in enabling the development of scalable clusters. In the two years following the first U-Net publications, a consortium led by Intel, Compaq and Microsoft developed the *Virtual Interface Architecture*, which was largely based on the experiences with U-Net, but also incorporated some of the direct memory transfer technology developed for the *VMMC2* architecture of the *Shrimp* project at Princeton. A description of the evolution of these research prototypes into the Virtual Interface Architecture appeared in the November 1998 issue of IEEE Computer and can be found in chapter 3.

User-level communication in production use

In 1998 commercial implementations of direct-user-level network interfaces and interconnects based on the VI architecture started to find their way into production clusters. Even though the architecture was able to meet the expectations, its success was not as wide-spread as expected. The main obstacles were difficulties in the transition of legacy cluster applications to a new network architecture combined with the lack of end-to-end control strategies. In VIA, just like in U-Net, the data and control transfer was separated and this was not a model many application developers were familiar with. Already quite quickly after the first introduction of VIA based hardware, network support libraries started to emerge that provided a more traditional programming interface. This ultimately resulted in the development of *Winsock-Direct*, an emulation of the socket interface for user-level communication. The emulation was a disaster from the performance point of view, but it was deemed “good-enough” as it helped the transition of legacy applications to VIA based platforms.

The production use of the new clusters showed a major departure from the communication patterns that were normally seen in scientific computing clusters. In the traditional parallel computing world there is a high level of control where applications are being placed and what phases the communication goes through. This is essential for achieving optimal performance. In the production enterprise clusters there is hardly any control over which applications are active at which nodes, and the internal communication is often triggered by events external to the cluster. This results in communication patterns that are very hard to predict, which puts a heavy burden on the interconnects to manage the load in the network fairly.

The separation of data and control in the VIA interface removed the ability of the communication architecture to provide end-to-end back-pressure to manage competing data streams. In chapter 4 detailed experiments are described which I performed to investigate how effective the use of flow-control feedback to network adapters is, when one cannot control the processes generating and receiving the data streams. The experiments were performed at one of the first large production VIA installations, a 256 processor cluster using a multi-stage Giganet interconnect, consisting of 40 switches organized in a fat tree. The results of the experiments were presented at the 2000 IEEE Hot Interconnects conference.

The future of user-level communication

Although the adoption of user-level communication by application developers has been slow, major system services have been adapted to directly use the network interfaces. For example the large commercial database systems, as well as transaction services, e-mail servers, cluster management systems and middle-tier application servers have been redesigned to make use of VIA directly. Other systems services such as the *Direct Access File System* (DAFS), are still under development.

A related area that emerged soon after the first VIA devices were delivered was that of direct user-level access to network storage devices. Fiber-Channel is the dominant network technology used to access the storage devices and improved performance can be established through transferring data directly in and from user space. The architecture necessary to support this is simpler than the work that was needed to build general communication architectures such as U-Net and VIA. Most of the operating systems provide block IO interfaces that are already designed to transfer data directly to and from user-provided buffers through DMA. The resulting direct user-level IO architectures exploit this behavior.

The drive in the storage community to use standard internet protocols for access to the networked storage has resulted in initiatives such as *iSCSI*. This push combined with the availability of hardware TCP implementation will ensure that direct user-level access to data transferred over standard internet protocols will become available.

At the computer architecture level U-Net and VIA have also had significant impact as they were used as templates for the interface model of the Infiniband architecture. *Infiniband* is a specification for the development of a unified high-performance IO architecture, for which the first devices will become generally available in 2003. Next to providing the common concepts such as direct user-level access, remote DMA and a queue based interface, Infiniband extends the work done in U-Net and similar systems with atomic remote *compare&swap* and *fetch&add* operations, and with extended memory operation to reduce the cost of repeated memory registrations.

The work done in U-Net continues to impact the way new network, storage and IO devices are developed. A major lesson learned from the transition of U-Net into VIA and the wide-spread deployment of the architecture is that these advances in hardware and interface technologies at the architecture level do not guarantee successful acceptation of the technology by application developers. There is still significant research to be done to develop distributed systems support that can exploit the user-level communication paradigms and provide the server application developer with solid abstractions for building high-performance applications.

Chapter 2

U-Net: A User-Level Network Interface for Parallel and Distributed Computing

The U-Net communication architecture provides processes with a virtual view of a network interface to enable user- level access to high-speed communication devices. The architecture, implemented on standard workstations using off- the-shelf ATM communication hardware, removes the kernel from the communication path, while still providing full protection.

The model presented by U-Net allows for the construction of protocols at user level whose performance is only limited by the capabilities of the network. The architecture is extremely flexible in the sense that traditional protocols like TCP and UDP, as well as novel abstractions like Active Messages can be implemented efficiently. A U-Net prototype on an 8-node ATM cluster of standard workstations offers 65 microseconds round-trip latency and 15 Mbytes/sec bandwidth. It achieves TCP performance at maximum network bandwidth and demonstrates performance equivalent to Meiko CS-2 and TMC CM-5 supercomputers on a set of Split-C benchmarks.

2.1 Introduction

The increased availability of high-speed local area networks has shifted the bottleneck in local-area communication from the limited bandwidth of network fabrics to the software path traversed by messages at the sending and receiving ends. In particular, in a traditional Unix networking architecture, the path taken by messages through the kernel involves several copies and crosses multiple levels of abstraction between the device driver and the user application. The resulting processing overheads limit the peak communication bandwidth and cause high end-to-end message latencies. The effect is that users who upgrade from Ethernet to a faster network fail to observe an application speed-up commensurate with the improvement in raw network performance. A solution to this situation seems to elude vendors to a large degree because many fail to recognize the importance of per-message overhead and concentrate on peak bandwidths of long data streams instead. While this may be justifiable for a few applications such as video playback, most applications use relatively small messages and rely heavily on quick round-trip requests and replies. The increased use of techniques such as distributed shared memory, remote procedure calls, remote object-oriented method invocations, and distributed cooperative file caches will further increase the importance of low round-trip latencies and of high bandwidth at the low-latency point.

Many new application domains could benefit not only from higher network performance but also from a more flexible interface to the network. By placing all protocol processing into the kernel, the traditional networking architecture cannot easily support new protocols or new message send/receive interfaces. Integrating application specific information into protocol processing allows for higher efficiency and greater flexibility in protocol cost management. For example, the transmission of MPEG compressed video streams can greatly benefit from customized retransmission protocols, which embody knowledge of the real-time demands as well as the interdependencies among video frames [95]. Other applications can avoid copying message data by sending straight out of data structures. Being able to accommodate such application specific knowledge into the communication protocols becomes more and more important in order to be able to efficiently utilize the network and to couple the communication and the computation effectively.

One of the most promising techniques to improve both the performance and the flexibility of networking layer performance on workstation-class machines is to move parts of the protocol processing into user space. This research argues that in fact the entire protocol stack should be placed at user level and that the operating system and hardware should allow protected user-level access directly to the network. The goal is to remove the kernel completely from the critical path and to allow the communication layers used by each process to be tailored to its demands. The key issues that arise are

- multiplexing the network among processes,
- providing protection such that processes using the network cannot interfere with each other,

- managing limited communication resources without the aid of a kernel path, and
- designing an efficient yet versatile programming interface to the network.

Some of these issues have been solved in parallel machines such as in the Thinking Machines CM-5, the Meiko CS-2, and the IBM SP-2, all of which allow user-level access to the network. However, all these machines have a custom network and network interface, and they usually restrict the degree or form of multiprogramming permitted on each node. This implies that the techniques developed in these designs cannot be applied to workstation clusters directly. This chapter describes the U-Net architecture for user-level communication on an off-the-shelf hardware platform (SPARCStations with Fore Systems ATM interfaces) running a standard operating system (SunOS 4.1.3). The communication architecture virtualizes the network device so that each process has the illusion of owning the interface to the network. Protection is assured through kernel control of channel set-up and tear-down. The U-Net architecture is able to support both legacy protocols and novel networking abstractions: TCP and UDP as well as Active Messages are implemented and exhibit performance that is only limited by the processing capabilities of the network interface. Using Split-C, a state-of-the-art parallel language, the performance of seven benchmark programs on an ATM cluster of standard workstations rivals that of parallel machines. In all cases U-Net was able to expose the full potential of the ATM network by saturating the 140Mbits/sec fiber, using either traditional networking protocols or advanced parallel computing communication layers.

The major contributions of this research are to propose a simple user-level communication architecture (Sections 2.2 and 2.3) which is independent of the network interface hardware (i.e., it allows many hardware implementations), to describe two high-performance implementations on standard workstations (Section 2.4), and to evaluate its performance characteristics for communication in parallel programs (Sections 2.5 and 2.6) as well as for traditional protocols from the IP suite (Section 2.7). While other researchers have proposed user-level network interfaces independently, this is the first presentation of a full system which does not require custom hardware or OS modification and which supports traditional networking protocols as well as state of the art parallel language implementations. Since it exclusively uses off-the-shelf components, the system presented here establishes a baseline to which more radical proposals that include custom hardware or new OS architectures must be compared to.

2.2 Motivation and related work

The U-Net architecture focuses on reducing the processing overhead required to send and receive messages as well as on providing flexible access to the lowest layer of the network. The intent is three-fold:

- provide low-latency communication in local area settings,
- exploit the full network bandwidth even with small messages, and
- facilitate the use of novel communication protocols

2.2.1 The importance of low communication latencies

The latency of communication is mainly composed of *processing overhead* and network latency (time-of-flight). The term processing overhead is used here to refer to the time spent by the processor in handling messages at the sending and receiving ends. This may include buffer management, message copies, checksumming, flow-control handling, interrupt overhead, as well as controlling the network interface. Separating this overhead from the *network latency* distinguishes the costs stemming from the network fabric technology from those due to the networking software layers.

Recent advances in network fabric technology have dramatically improved network bandwidth while the processing overheads have not been affected nearly as much. The effect is that for large messages, the *end-to-end latency*—the time from the source application executing "send" to the time the destination application receiving the message—is dominated by the transmission time and the new networks offer a net improvement. For small messages in local area communication, however, the processing overheads dominate and the improvement in transmission time is less significant in comparison. In wide area networks the speed of light eventually becomes the dominant latency component and while reducing the overhead does not significantly affect latency it may well improve throughput.

U-Net places a strong emphasis on achieving low communication overheads because small messages are becoming increasingly important in many applications. For example, in distributed systems:

- Object-oriented technology is finding widespread adoption and is naturally extended across the network by allowing the transfer of objects and the remote execution of methods (e.g., CORBA and the many C++ extensions). Objects are generally small relative to the message sizes required for high bandwidth (around

100 bytes vs. several Kbytes) and thus communication performance suffers unless message overhead is low.

- The electronic workplace relies heavily on sets of complex distributed services, which are intended to be transparent to the user. The majority of such service invocations are requests to simple database servers that implement mechanisms like object naming, object location, authentication, protection, etc. The message size seen in these systems range from 20-80 bytes for the requests and the responses generally can be found in the range of 40-200 bytes.

- To limit the network traversal of larger distributed objects, caching techniques have become a fundamental part of most modern distributed systems. Keeping the copies consistent introduces a large number of small coherence messages. The round-trip times are important as the requestor is usually blocked until the synchronization is achieved.

- Software fault-tolerance algorithms and group communication tools often require multi-round protocols, the performance of which is latency-limited. High processing overheads resulting in high communication latencies prevent such protocols from being used today in process-control applications, financial trading systems, or multimedia groupware applications.

- Without projecting into the future, existing more general systems can benefit substantially as well:
    - Numerous client/server architectures are based on a RPC style of interaction. By drastically improving the communication latency for requests, responses and their acknowledgments, many systems may see significant performance improvements.
    - Although remote file systems are often categorized as bulk transfer systems, they depend heavily on the performance of small messages. A weeklong trace of all NFS traffic to the departmental CS fileserver at UC Berkeley has shown that the vast majority of the messages is under 200 bytes in size and that these messages account for roughly half the bits sent [2].

Finally, many researchers propose to use networks of workstations to provide the resources for compute intensive parallel applications. In order for this to become feasible, the communication costs across LANs must reduce by more than an order of magnitude to be comparable to those on modern parallel machines.

2.2.2 The importance of small-message bandwidth

The communication bandwidth of a system is often measured by sending a virtually infinite stream from one node to another. While this may be representative of a few applications, the demand for high bandwidths when sending many small messages (e.g., a few hundred bytes) is increasing due to the same trends that demand low latencies. U-Net specifically targets this segment of the network traffic and attempts to provide full network bandwidth with as small messages as possible, mainly by reducing the per-message overheads.

Reducing the minimal message size at which full bandwidth can be achieved may also benefit reliable data stream protocols like TCP that have buffer requirements that are directly proportional to the round-trip end-to-end latency. For example the TCP window size is the product of the network bandwidth and the round-trip time. Achieving low-latency in local area networks will keep the buffer consumption within reason and thus make it feasible to achieve maximal bandwidth at low cost.

2.2.3 Communication protocol and interface flexibility

In traditional UNIX networking architectures the protocol stacks are implemented as part of the kernel. This makes it difficult to experiment with new protocols and efficiently support dedicated protocols that deal with application specific requirements. Although one could easily design these protocols to make use of a datagram primitive offered by the kernel (like UDP or raw IP), doing so efficiently without adding the new protocol to the kernel stack is not possible. The lack of support for the integration of kernel and application buffer management introduces high processing overheads, which especially affect reliable protocols that need to keep data around for retransmission. In particular, without shared buffer management reference count mechanisms cannot be used to lower the copy and application/kernel transfer overheads. For example, a kernel-based implementation of a reliable transport protocol like TCP can use reference counts to prevent the network device driver from releasing network buffers that must remain available for possible retransmission. Such an optimization is not available if an application specific reliable protocol is implemented in user space and has to use UDP as transport mechanism.

By removing the communication subsystem's boundary with the application-specific protocols, new protocol design techniques, such as Application Level Framing [18,41] and Integrated Layer Processing [1,15,18], can be applied and more efficient protocols produced. Compiler assisted protocol development can achieve maximum

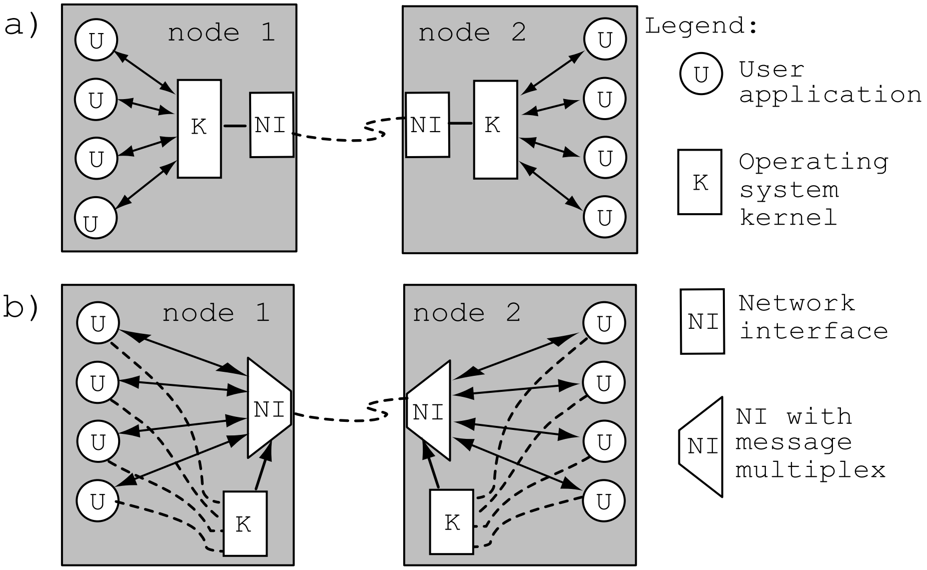


Figure 2.1. The traditional networking architecture (a) places the kernel in the path of the communication. The U-Net architecture (b) only uses a simple multiplexing/ demultiplexing agent - that can be implemented in hardware - in the data communication path and uses the kernel only for set-up.

optimization if all protocols are compiled together instead of only a small subset of application specific protocols.

In more specialized settings a tight coupling between the communication protocol and the application can yield even higher savings. For example, in a high-level language supporting a form of blocking RPC no copy need to be made in case a retransmission is required, as the high-level semantics of the RPC guarantee that the source data remains unmodified until the RPC completes successfully. Thus the address of a large RPC argument may well be passed down directly to the network interface's DMA engine.

Another example is that at the moment a process requests data from a remote node it may pre-allocate memory for the reply. When the response arrives, the data can be transferred directly into its final position without the allocation of intermediate buffers or any intermediate copies.

Taking advantage of the above techniques is becoming a key element in reducing the overhead of communication and can only be done if applications have direct access to the network interface.

2.2.4 Towards a new networking architecture

A new abstraction for high-performance communication is required to deliver the promise of low-latency, high-bandwidth communication to the applications on standard workstations using off-the-shelf networks. The central idea in U-Net is to simply remove the kernel from the critical path of sending and receiving messages. This eliminates the system call overhead, and more importantly, offers the opportunity to streamline the buffer management, which can now be performed at user-level. As several research projects have pointed out, eliminating the kernel from the send and receive paths requires that some form of a message multiplexing and demultiplexing device (in hardware or in software) is introduced for the purpose of enforcing protection boundaries.

The approach proposed in this research is to incorporate this mux/demux directly into the network interface (NI), as depicted in Figure 2.1, and to move all buffer management and protocol processing to user-level. This, in essence, virtualizes the NI and provides each process the illusion of owning the interface to the network. Such an approach raises the issues of selecting a good virtual NI abstraction to present to processes, of providing support for legacy protocols side-by-side with next generation parallel languages, and of enforcing protection without kernel intervention on every message.

2.2.5 Related work

Some of the issues surrounding user-level network interface access have been studied in the past. For the Mach3 operating system a combination of a powerful message demultiplexer in the microkernel, and a user-level implementation of the TCP/IP protocol suite solved the network performance problems that arose when the Unix single OS-Server was responsible for all network communication. The performance achieved is roughly the same as that of a monolithic BSD system [62].

More recently, the *application device channel* abstraction, developed at the University of Arizona, provides application programs with direct access to the experimental Osiris ATM board [30] used in the Aurora Gigabit testbed. Other techniques that are developed for the Osiris board to reduce the processing overhead are the pathfinder multiplexer [4], which is implemented in hardware and the *fbufs* cross-domain buffer management [29].

At HP Bristol a mechanism has been developed for the Jetstream LAN [34] where applications can reserve buffer pools on the Afterburner [26] board. When data arrives on a virtual circuit associated with an application, data is transferred directly into the correct pool. However, the application cannot access these buffers directly: it is always forced to go through the kernel with a copy operation to retrieve the data or provide data for sending. Only the kernel-based protocols could be made aware of the buffer pools and exploit them fully.

In the parallel computing community some machines (e.g., Thinking Machines CM-5, Meiko CS-2, IBM SP-2, Cray T3D) provide user-level access to the network, but the solutions rely on custom hardware and are somewhat constrained to the controlled environment of a multiprocessor. On the other hand, given that these parallel machines resemble clusters of workstations ever more closely, it is reasonable to expect that some of the concepts developed in these designs can indeed be transferred to workstations.

Successive simplifications and generalizations of shared memory is leading to a slightly different type of solution in which the network can be accessed indirectly through memory accesses. Shrimp [12] uses custom NIs to allow processes to establish channels connecting virtual memory pages on two nodes such that data written into a page on one side gets propagated automatically to the other side. Thekkath [98] proposes a memory-based network access model that separates the flow of control from the data flow. The remote memory operations have been implemented by emulating unused opcodes in the MIPS instruction set. While the use of a shared memory abstraction allows a reduction of the communication overheads, it is not clear how to efficiently support legacy protocols, long data streams, or remote procedure call.

2.2.6 U-Net design goals

Experience with network interfaces in parallel machines made it clear that providing user-level access to the network in U-Net is the best avenue towards offering communication latencies and bandwidths that are mainly limited by the network fabric and that, at the same time, offer full flexibility in protocol design and in the integration of protocol, buffering, and appropriate higher communication layers. The many efforts in developing fast implementations of TCP and other internetworking

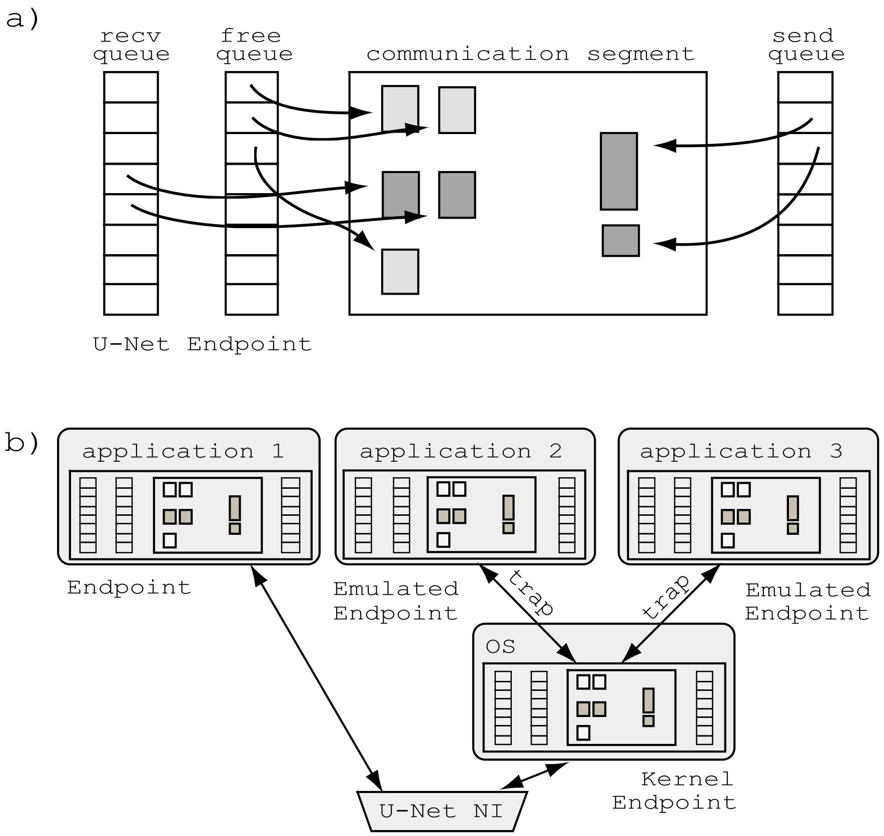


Figure 2.2. U-Net building blocks.

a) *Endpoints* serve as an application's handle into the network, *communication segments* are regions of memory that hold message data, and message queues (*send/recv/free* queues) hold descriptors for messages that are to be sent or that have been received.

b) Regular endpoints are serviced by the U-Net network interface directly. Emulated endpoints are serviced by the kernel and consume no additional network interface resources but cannot offer the same level of performance.

protocols clearly affirm the relevance of these protocols in high-performance networking and thus any new network interface proposal must be able to support these protocols effectively (which is typically not the case in parallel machines, for example).

The three aspects that set U-Net apart from the proposals discussed above are:

- the focus on low latency and high bandwidth using small messages,
- the emphasis on protocol design and integration flexibility, and

- the desire to meet the first two goals on widely available standard workstations using off-the-shelf communication hardware.

2.3 The user-level network interface architecture

The U-Net user-level network interface architecture virtualizes the interface in such a way that a combination of operating system and hardware mechanisms can provide every process[1] the illusion of owning the interface to the network. Depending on the sophistication of the actual hardware, the U-Net components manipulated by a process may correspond to real hardware in the NI, to memory locations that are interpreted by the OS, or to a combination of the two. The role of U-Net is limited to multiplexing the actual NI among all processes accessing the network and enforcing protection boundaries as well as resource consumption limits. In particular, a process has control over both the contents of each message and the management of send and receive resources, such as buffers.

2.3.1 Sending and receiving messages

The U-Net architecture is composed of three main building blocks, shown in Figure 2.2: *endpoints* serve as an application’s handle into the network and contain *communication segments* which are regions of memory that hold message data, and *message queues* which hold descriptors for messages that are to be sent or that have been received. Each process that wishes to access the network first creates one or more endpoints, then associates a communication segment and a set of *send*, *receive*, and *free* message queues with each endpoint.

To send a message, a user process composes the data in the communication segment and pushes a descriptor for the message onto the send queue. At that point, the network interface is expected to pick the message up and insert it into the network. If the network is backed-up, the network interface will simply leave the descriptor in the queue and eventually exert backpressure to the user process when the queue becomes full. The NI provides a mechanism to indicate whether a message in the queue has been injected into the network, typically by setting a flag in the descriptor; this indicates that the associated send buffer can be reused.

Incoming messages are demultiplexed by U-Net based on their destination: the data is transferred into the appropriate communication segment and a message descriptor

[1] The terms “process” and “application” are used interchangeably to refer to arbitrary unprivileged UNIX processes.

is pushed onto the corresponding receive queue. The receive model supported by U-Net is either polling or event driven: the process can periodically check the status of the receive queue, it can block waiting for the next message to arrive (using a UNIX select call), or it can register an upcall[2] with U-Net. The upcall is used by U-Net to signal that the state of the receive queue satisfies a specific condition. The two conditions currently supported are: the receive queue is non-empty and the receive queue is almost full. The first one allows event driven reception while the second allows processes to be notified before the receive queue overflows. U-Net does not specify the nature of the upcall which could be a UNIX signal handler, a thread, or a user-level interrupt handler.

In order to amortize the cost of an upcall over the reception of several messages it is important that a U-Net implementation allows all messages pending in the receive queue to be consumed in a single upcall. Furthermore, a process must be able to disable upcalls cheaply in order to form critical sections of code that are atomic relative to message reception.

2.3.2 Multiplexing and demultiplexing messages

U-Net uses a tag in each incoming message to determine its destination endpoint and thus the appropriate communication segment for the data and message queue for the descriptor. The exact form of this message tag depends on the network substrate; for example, in an ATM network the ATM virtual channel identifiers (VCIs) may be used. In any case, a process registers these tags with U-Net by creating communication channels: on outgoing messages the channel identifier is used to place the correct tag into the message (as well as possibly the destination address or route) and on incoming messages the tag is mapped into a channel identifier to signal the origin of the message to the application.

U-Net's notion of a message tag is similar to the idea used in parallel machines of including a parallel-process id in the header of messages. The message tag used in U-Net is more general, however, in that it allows communication between arbitrary processes, whereas a parallel-process id tag only serves communication within a parallel program running in a closed environment.

An operating system service needs to assist the application in determining the correct tag to use based on a specification of the destination process and the route between

[2] The term "upcall" is used in a very general sense to refer to a mechanism which allows U-Net to signal an asynchronous event to the application.

the two nodes. The operating system service will assist in route discovery, switch-path setup and other (signalling) tasks that are specific for the network technology used. The service will also perform the necessary authentication and authorization checks to ensure that the application is allowed access to the specific network resources and that there are no conflicts with other applications. After the path to the peer has been determined and the request has passed the security constraints, the resulting tag will be registered with U-Net such that the latter can perform its message multiplexing/demultiplexing function. A channel identifier is returned to the requesting application to identify the communication channel to the destination.

Endpoints and communication channels together allow U-Net to enforce protection boundaries among multiple processes accessing the network and, depending on how routes are allocated, may allow it to extend these boundaries across the network. This is achieved using two mechanisms:

- endpoints, communication segments, and message queues are only accessible by the owning process,
- outgoing messages are tagged with the originating endpoint address and incoming messages are demultiplexed by U-Net and only delivered to the correct destination endpoint.

Thus an application cannot interfere with the communication channels of another application on the same host. In addition, if the set-up of routes is carefully controlled by the collection of operating systems in a cluster of hosts, then this protection can be extended across the network such that no application can directly interfere with communication streams between other parties.

2.3.3 Zero-copy vs. true zero-copy

U-Net attempts to support a “true zero copy” architecture in which data can be sent directly out of the application data structures without intermediate buffering and where the NI can transfer arriving data directly into user-level data structures as well. In consideration of current limitations on I/O bus addressing and on NI functionality, the U-Net architecture specifies two levels of sophistication: a *base-level* which requires an intermediate copy into a networking buffer and corresponds to what is generally referred-to as zero copy, and a *direct-access* U-Net which supports true zero copy without any intermediate buffering.

The base-level U-Net architecture matches the operation of existing network adapters by providing a reception model based on a queue of free buffers that are filled by U-Net as messages arrive. It also regards communication segments as a limited resource and places an upper bound on their size such that it is not feasible to regard communication segments as memory regions in which general data structures can be placed. This means that for sending, each message must be constructed in a buffer in the communication segment and on reception data is deposited in a similar buffer. This corresponds to what is generally called "zero-copy", but which in truth represents one copy, namely between the application's data structures and a buffer in the communication segment.[3]

Direct-access U-Net supports true zero copy protocols by allowing communication segments to span the entire process address space and by letting the sender specify an offset within the destination communication segment at which the message data is to be deposited directly by the NI.

The U-Net implementations described here support the base- level architecture because the hardware available does not support the memory mapping required for the direct-access architecture. In addition, the bandwidth of the ATM network used does not warrant the enhancement because the copy overhead is not a dominant cost.

2.3.4 Base-level U-Net architecture

The base-level U-Net architecture supports a queue-based interface to the network which stages messages in a limited-size communication segment on their way between application data structures and the network. The communication segments are allocated to buffer message data and are typically pinned to physical memory. In the base-level U-Net architecture send and receive queues hold descriptors with information about the destination, respectively origin, endpoint addresses of messages, their length, as well as offsets within the communication segment to the data. Free queues hold descriptors for free buffers that are made available to the network interface for storing arriving messages.

[3] True zero copy is achieved with base-level U-Net when there is no need for the application to copy the information received to a data structure for later reference. In that case data can be accessed in the buffers and the application can take action based on this information without the need for a copy operation. A simple example of this is the reception of acknowledgment messages that are used to update some counters but do not need to be copied into longer term storage.

The management of send buffers is entirely up to the process: the U-Net architecture does not place any constraints on the size or number of buffers nor on the allocation policy used. The only restrictions are that buffers lie within the communication segment and that they be properly aligned for the requirements of the network interface (e.g., to allow DMA transfers). The process also provides receive buffers explicitly to the NI via the free queue but it cannot control the order in which these buffers are filled with incoming data.

As an optimization for small messages—which are used heavily as control messages in protocol implementation—the send and receive queues may hold entire small messages in descriptors (i.e., instead of pointers to the data). This avoids buffer management overheads and can improve the round-trip latency substantially. The size of these small messages is implementation dependent and typically reflects the properties of the underlying network.

2.3.5 Kernel emulation of U-Net

Communication segments and message queues are generally scarce resources and it is often impractical to provide every process with U-Net endpoints. Furthermore many applications (such as telnet) do not really benefit from that level of performance. Yet, for software engineering reasons it may well be desirable to use a single interface to the network across all applications. The solution to this dilemma is to provide applications with kernel-emulated U- Net endpoints. To the application these emulated endpoints look just like regular ones, except that the performance characteristics are quite different because the kernel multiplexes all of them onto a single real endpoint.

2.3.6 Direct-Access U-Net architecture

Direct-access U-Net is a strict superset of the base-level architecture. It allows communication segments to span the entire address space of a process and it allows senders to specify an offset in the destination communication segment at which the message data is to be deposited. This capability allows message data to be transferred directly into application data structures without any intermediate copy into a buffer. While this form of communication requires quite some synchronization between communicating processes, parallel language implementations, such as Split-C, can take advantage of this facility.

Operation	Time (μs)
1-way send and receiver across the switch (at trap level)	21
Send overhead (AAL5)	7
Receive overhead (AAL5)	5
Total (one way)	33

Table 2.1. Cost breakup for a single-roundtrip (AAL5).

The main problem with the direct-access U-Net architecture is that it is difficult to implement on current workstation hardware: the NI must essentially contain an MMU that is kept consistent with the main processor's and the NI must be able to handle incoming messages which are destined to an unmapped virtual memory page. Thus, in essence, it requires (i) the NI to include some form of memory mapping hardware, (ii) all of (local) physical memory to be accessible from the NI, and (iii) page faults on message arrival to be handled appropriately.

At a more basic hardware level, the limited number of address lines on most I/O buses makes it impossible for an NI to access all of physical memory such that even with an on-board MMU it is very difficult to support arbitrary-sized communication segments.

2.4 Two U-Net implementations

The U-Net architecture has been implemented on SPARCstations running SunOS 4.1.3 and using two generations of Fore Systems ATM interfaces. The first implementation uses the Fore SBA- 100 interface and is very similar to an Active Messages implementation on that same hardware described elsewhere [36]. The second implementation uses the newer Fore SBA-200 interface and reprograms the on-board i960 processor to implement U-Net directly. Both implementations transport messages in AAL5 packets and take advantage of the ATM virtual channel identifiers in that all communication between two endpoints is associated with a transmit/receive VCI pair[4].

2.4.1 U-Net using the SBA-100

The Fore Systems SBA-100 interface operates using programmed I/O to store cells into a 36-cell deep output FIFO and to retrieve incoming cells from a 292-cell deep

[4] ATM is a connection-oriented network that uses virtual channel identifiers (VCIs) to name one-way connections.

input FIFO. The only function performed in hardware beyond serializing cells onto the fiber is ATM header CRC calculation. In particular, no DMA, no payload CRC calculation[5], and no segmentation and reassembly of multi-cell packets are supported by the interface. The simplicity of the hardware requires the U-Net architecture to be implemented in the kernel by providing emulated U-Net endpoints to the applications as described in the section on kernel emulation.

The implementation consists of a loadable device driver and a user-level library implementing the AAL5 segmentation and reassembly (SAR) layer. Fast traps into the kernel are used to send and receive individual ATM cells: each is carefully crafted in assembly language and is quite small (28 and 43 instructions for the send and receive traps, respectively).

The implementation was evaluated on two 60Mhz SPARCstation-20s running SunOS 4.1.3 and equipped with SBA-100 interfaces. The ATM network consists of 140Mbit/s TAXI fibers leading to a Fore Systems ASX-200 switch. The end-to-end round trip time of a single-cell message is 66µs. A consequence of the lack of hardware to compute the AAL5 CRC is that 33% of the send overhead and 40% of the receive overhead in the AAL5 processing is due to CRC computation. The cost breakup is shown in Table 2.1. Given the send and receive overheads, the bandwidth is limited to 6.8MBytes/s for packets of 1KBytes.

2.4.2 U-Net using the SBA-200

The second generation of ATM network interfaces produced by Fore Systems, the SBA-200, is substantially more sophisticated than the SBA-100 and includes an on-board processor to accelerate segmentation and reassembly of packets as well as to transfer data to/from host memory using DMA. This processor is controlled by firmware which is downloaded into the on-board RAM by the host. The U-Net implementation described here uses custom firmware to implement the base-level architecture directly on the SBA- 200.

The SBA-200 consists of a 25Mhz Intel i960 processor, 256Kbytes of memory, a DMA-capable I/O bus (Sbus) interface, a simple FIFO interface to the ATM fiber (similar to the SBA-100), and an AAL5 CRC generator. The host processor can map the SBA-200 memory into its address space in order to communicate with the i960 during operation.

[5] The card calculates the AAL3/4 checksum over the payload but not the AAL5 CRC required here.

The experimental set-up used consists of five 60Mhz SPARCStation-20 and three 50Mhz SPARCStation-10 workstations connected to a Fore Systems ASX-200 ATM switch with 140Mbit/s TAXI fiber links.

Fore firmware operation and performance

The complete redesign of the SBA-200 firmware for the U-Net implementation was motivated by an analysis of Fore's original firmware which showed poor performance. The apparent rationale underlying the design of Fore's firmware is to off-load the specifics of the ATM adaptation layer processing from the host processor as much as possible. The kernel-firmware interface is patterned after the data structures used for managing BSD mbufs and System V streams bufs. It allows the i960 to traverse these data structures using DMA in order to determine the location of message data, and then to move it into or out of the network rather autonomously.

The performance potential of Fore's firmware was evaluated using a test program which maps the kernel-firmware interface data structures into user space and manipulates them directly to send raw AAL5 PDUs over the network. The measured round-trip time was approximately 160µs while the maximum bandwidth achieved using 4Kbyte packets was 13Mbytes/sec. This performance is rather discouraging: the round-trip time is almost 3 times larger than using the much simpler and cheaper SBA-100 interface, and the bandwidth for reasonable sized packets falls short of the 15.2Mbytes/sec peak fiber bandwidth.

A more detailed analysis showed that the poor performance can mainly be attributed to the complexity of the kernel-firmware interface. The message data structures are more complex than necessary and having the i960 follow linked data structures on the host using DMA incurs high latencies. Finally, the host processor is much faster than the i960 and so off-loading can easily backfire.

U-Net firmware

The base-level U-Net implementation for the SBA-200 modifies the firmware to add a new U-Net compatible interface[6]. The main design considerations for the new firmware were to virtualize the host-i960 interface such that multiple user processes

[6] For software engineering reasons, the new firmware's functionality is a strict superset of Fore's such that the traditional networking layers can still function while new applications can use the faster U-Net.

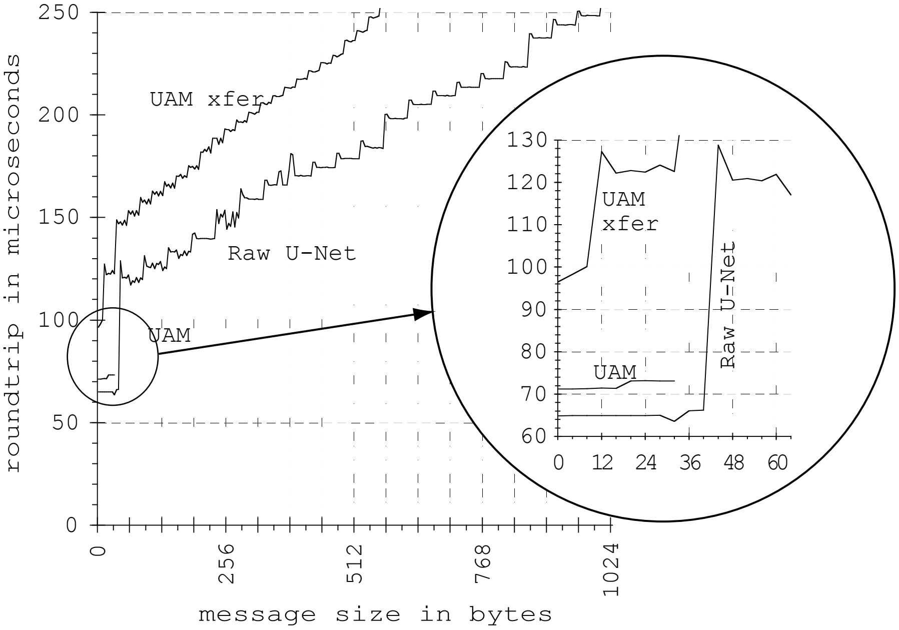


Figure 2.3. U-Net round-trip times as a function of message size. The *Raw U-Net* graph shows the round-trip times for a simple ping-pong benchmark using the U-Net interface directly. The small graph highlights the performance on small messages. The *UAM* line measures the performance of U-Net Active Messages using reliable single-cell requests and replies whereas *UAM xfer* uses reliable block transfers of arbitrary size.

can communicate with the i960 concurrently, and to minimize the number of host and i960 accesses across the I/O bus.

The new host-i960 interface reflects the base-level U-Net architecture directly. The i960 maintains a data structure holding the protection information for all open endpoints. Communication segments are pinned to physical memory and mapped into the i960's DMA space, receive queues are similarly allocated such that the host can poll them without crossing the I/O bus, while send and free queues are actually placed in SBA-200 memory and mapped into user-space such that the i960 can poll these queues without DMA transfers.

The control interface to U-Net on the i960 consists of a single i960-resident command queue that is only accessible from the kernel. Processes use the system call interface to the device driver that implements the kernel resident part of U-Net. This driver assists in providing protection by validating requests for the creation of communication segments and related endpoints, and by providing a secure interface

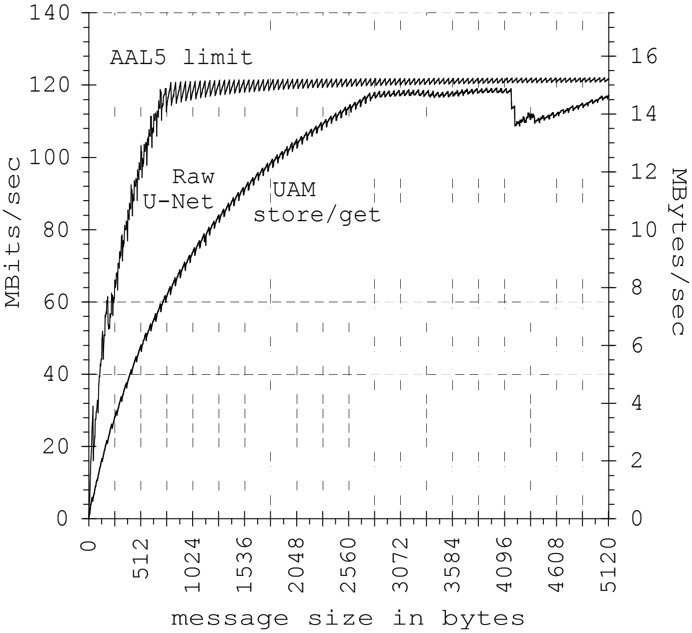


Figure 2.4. U-Net bandwidth as a function of message size. The *AAL-5* limit curve represents the theoretical peak bandwidth of the fiber (the sawtooths are caused by the quantization into 48-byte cells). The *Raw U-Net* measurement shows the bandwidth achievable using the U-Net interface directly, while *UAM store/get* demonstrate the performance of reliable U-Net Active Messages block transfers.

between the operating system service that manages the multiplexing tags and the U-Net channel registration with the i960. The tags used for the ATM network consist of a VCI pair that implements full duplex communication (ATM is a connection oriented network and requires explicit connection set-up even though U-Net itself is not connection oriented). The communication segments and message queues for distinct endpoints are disjoint and are only present in the address space of the process that creates the endpoint.

In order to send a single protocol data unit (PDU), the host uses a double word store to the i960-resident transmit queue to provide a pointer to a transmit buffer, the length of the packet and the channel identifier to the i960. Single cell packet sends are optimized in the firmware because many small messages are less than a cell in size. For larger sized messages, the host-i960 DMA uses three 32-byte burst transfers to fetch two cells at a time and computes the AAL5 CRC using special SBA-200 hardware.

To receive cells from the network, the i960 periodically polls the network input FIFO. Receiving single cell messages is special- cased to improve the round-trip latency for small messages. The single-cell messages are directly transferred into the next receive-queue entry which is large enough to hold the entire message—this avoids buffer allocation and extra DMA for the buffer pointers. Longer messages are transferred to fixed-size receive buffers whose offsets in the communication segment are pulled off the i960-resident free queue. When the last cell of the packet is received, the message descriptor containing the pointers to the buffers is DMA-ed into the next receive queue entry.

Performance

Figure 2.3 shows the round trip times for messages up to 1K bytes, i.e., the time for a message to go from one host to another via the switch and back. The round-trip time is 65µs for a one-cell message due to the optimization, which is rather low, but not quite at par with parallel machines, like the CM-5, where custom network interfaces placed on the memory bus (Mbus) allow round- trips in 12µs. Using a UNIX signal to indicate message arrival instead of polling adds approximately another 30µs on each end. Longer messages start at 120µs for 48 bytes and cost roughly an extra 6µs per additional cell (i.e., 48 bytes). Figure 2.4 shows the bandwidth over the raw base level U-Net interface in Mbytes/sec for message sizes varying from 4 bytes to 5Kbytes. It is clear from the graph that with packet sizes as low as 800 bytes, the fiber can be saturated.

Memory requirements

The current implementation pins pages used in communication segments down to physical memory and maps them into the SBA- 200’s DMA space. In addition, each endpoint has its own set of send, receive and free buffer queues, two of which reside on the i960 and are mapped to user-space. The number of distinct applications that can be run concurrently is therefore limited by the amount of memory that can be pinned down on the host, the size of the DMA address space and, the i960 memory size. Memory resource management is an important issue if access to the network interface is to be scalable. A reasonable approach would be to provide a mechanism by which the i960, in conjunction with the kernel, would provide some elementary memory management functions, which would allow dynamic allocation of the DMA address space to the communication segments of active user processes. The exact mechanism to achieve such an objective without compromising the efficiency and simplicity of the interface remains a challenging problem.

2.5 U-Net Active Messages implementation and performance

The U-Net Active Messages (UAM) layer is a prototype that conforms to the Generic Active Messages (GAM) 1.1 specification [24]. Active Messages is a mechanism that allows efficient overlapping of communication with computation in multiprocessors. Communication using Active Messages is in the form of requests and matching replies. An Active Message contains the address of a handler that gets called on receipt of the message followed by upto four words of arguments. The function of the handler is to pull the message out of the network and integrate it into the ongoing computation. A request message handler may or may not send a reply message. However, in order to prevent live-lock, a reply message handler cannot send another reply.

Generic Active Messages consists of a set of primitives that higher-level layers can use to initialize the GAM interface, send request and reply messages and perform block gets and stores. GAM provides reliable message delivery, which implies that a message that is sent will be delivered to the recipient barring network partitions, node crashes, or other catastrophic failures.

2.5.1 Active Messages implementation

The UAM implementation consists of a user level library that exports the GAM 1.1 interface and uses the U-Net interface. The library is rather simple and mainly performs the flow-control and retransmissions necessary to implement reliable delivery and the Active Messages-specific handler dispatch.

Flow Control Issues

In order to ensure reliable message delivery, UAM uses a window-based flow control protocol with a fixed window size (w). Every endpoint preallocates a total of 4w transmit and receive buffers for every endpoint it communicates with. This storage allows w requests and w replies to be kept in case retransmission is needed and it allows 2w request and reply messages to arrive without buffer overflow.

Request messages which do not generate a reply are explicitly acknowledged and a standard “go back N” retransmission mechanism is used to deal with lost requests or replies. The flow control implemented here is an end-to-end flow control mechanism which does not attempt to minimize message losses due to congestion in the network.

Sending and Receiving

To send a request message, UAM first processes any outstanding messages in the receive queue, drops a copy of the message to be sent into a pre-allocated transmit buffer and pushes a descriptor onto the send queue. If the send window is full, the sender polls for incoming messages until there is space in the send window or until a time-out occurs and all unacknowledged messages are retransmitted. The sending of reply messages or explicit acknowledgments is similar except that no flow-control window check is necessary.

The UAM layer receives messages by explicit polling. On message arrival, UAM loops through the receive queue, pulls the messages out of the receive buffers, dispatches the handlers, sends explicit acknowledgments where necessary, and frees the buffers and the receive queue entries.

2.5.2 Active Messages micro-benchmarks

Four different micro-benchmarks were run to determine the round trip times and transfer bandwidths for single cell messages as well as block transfers.

1. The single-cell round trip time was estimated by repeatedly sending a single cell request message with 0 to 32 bytes of data to a remote host specifying a handler that replies with an identical message. The measured round trip times are shown in Figure 2.3 and start at 71µs which suggests that the UAM overhead over raw U-Net is about 6µs. This includes the costs to send a request message, receive it, reply and receive the reply.

2. The block transfer round-trip time was measured similarly by sending messages of varying sizes back and forth between two hosts. Figure 2.3 shows that the time for an N-byte transfer is roughly 135µs + N*0.2µs. The per-byte cost is higher than for Raw U-Net because each one-way UAM transfer involves two copies (from the source data structure into a send buffer and from the receive buffer into the destination data structure).

3. The block store bandwidth was measured by repeatedly storing a block of a specified size to a remote node in a loop and measuring the total time taken. Figure 2.4 shows that the bandwidth reaches 80% of the AAL-5 limit with blocks

Machine	CPU speed	message overhead	round-trip latency	network bandwidth
CM-5	33 MHz Sparc-2	3 µs	12 µs	10 Mb/s
Meiko CS-2	40 MHz SuperSparc	11 µs	25 µs	39 Mb/s
U-Net ATM	50/60 Mhz SuperSparc	6 µs	71 µs	14 Mb/s

Table 2.2. Comparison of CM-5, Meiko CS-2, and U-Net ATM cluster computation and communication performance characteristics

of about 2Kbytes. The dip in performance at 4164 bytes is caused by the fact that UAM uses buffers holding 4160 bytes of data and thus additional processing time is required. The peak bandwidth at 4Kbytes is 14.8Mbytes/s.

The block get bandwidth was measured by sending a series of requests to a remote node to fetch a block of specified size and waiting until all blocks arrive. The block get performance is nearly identical to that of block stores.

2.5.3 Summary

The performance of Active Messages shows that the U-Net interface is well suited for building higher-level communication paradigms used by parallel languages and run-times. The main performance penalty of UAM over raw U-Net is due to the cost of implementing reliability and removing the restrictions of the communication segment size: UAM must send acknowledgment messages and it copies data into and out of buffers in the communication segment. For large transfers there is virtually no bandwidth loss due to the extra copies, but for small messages the extra overhead of the copies and the acknowledgments is noticeable.

Overall, the performance of UAM is so close to raw U-Net that using the raw interface is only worthwhile if control over every byte in the AAL-5 packets is required (e.g., for compatibility) or if significant benefits can be achieved by using customized retransmission protocols.

2.6 Split-C application benchmarks

Split-C [22] is a simple parallel extension to C for programming distributed memory machines using a global address space abstraction. It is implemented on top of U-Net Active Messages and is used here to demonstrate the impact of U-Net on applications written in a parallel language. A Split-C program comprises one thread of control per processor from a single code image and the threads interact through reads and writes

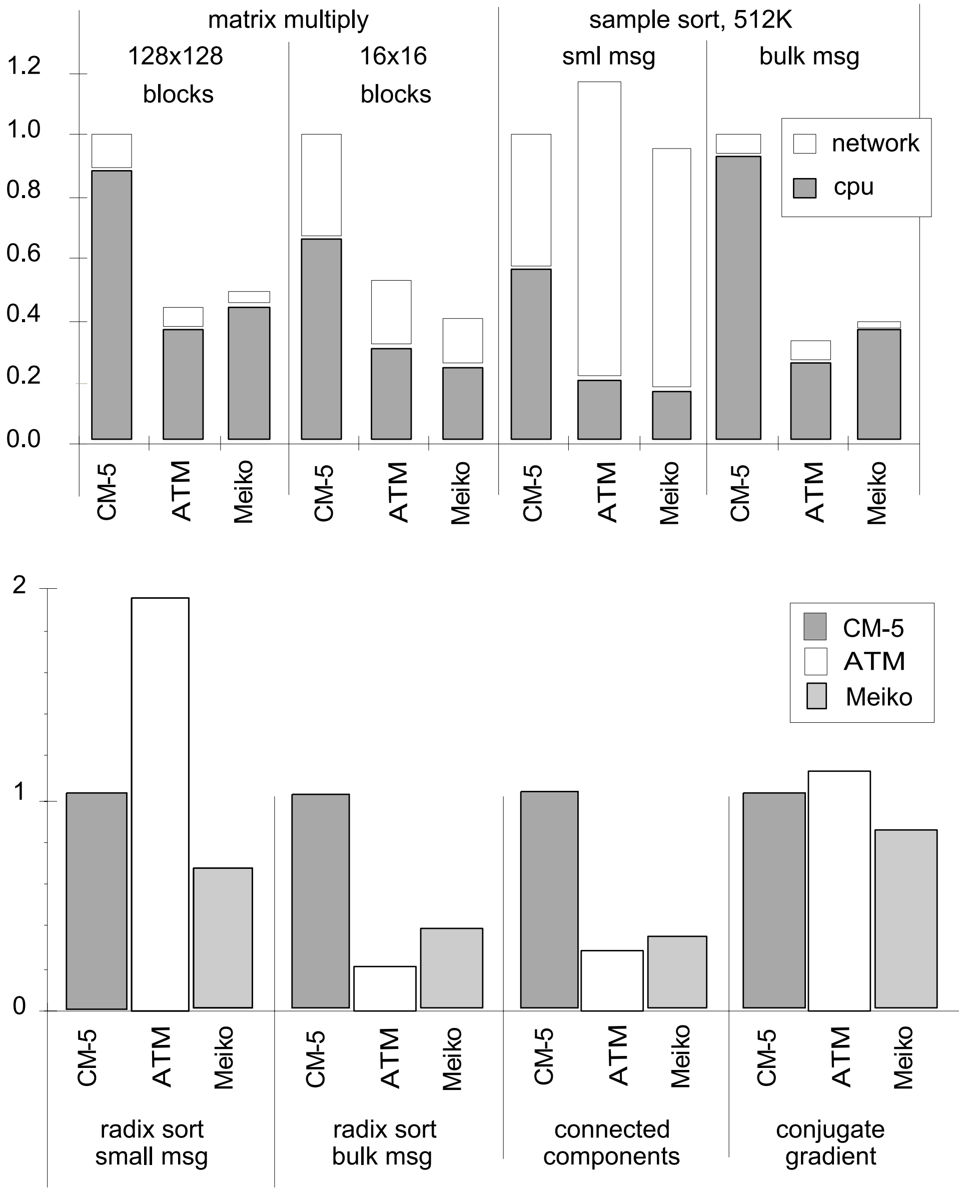


Figure 2.5. Comparison of seven Split-C benchmarks on the CM-5, the U-Net ATM cluster, and the Meiko CS-2. The execution times are normalized to the CM-5 and the computation/communication breakdown is shown for three applications.

on shared data. The type system distinguishes between local and global pointers such that the compiler can issue the appropriate calls to Active Messages whenever a global pointer is dereferenced. Thus, dereferencing a global pointer to a scalar variable turns into a request and reply Active Messages sequence exchange with the processor holding the data value. Split-C also provides bulk transfers which map into Active Message bulk gets and stores to amortize the overhead over a large data transfer.

Split-C has been implemented on the CM-5, Paragon, SP-1, Meiko CS-2, IBM SP-2, and Cray T3D supercomputers as well as over U-Net Active Messages. A small set of application benchmarks is used here to compare the U-Net version of Split-C to the CM-5 [22,35] and Meiko CS-2 [90] versions. This comparison is particularly interesting as the CM-5 and Meiko machines are easily characterized with respect to the U-Net ATM cluster as shown in Table 2.2: the CM-5's processors are slower than the Meiko's and the ATM cluster's, but its network has lower overheads and latencies. The CS-2 and the ATM cluster have very similar characteristics with a slight CPU edge for the cluster and a faster network for the CS-2.

The Split-C benchmark set used here is comprised of seven programs: a blocked matrix multiply [22], a sample sort optimized for small messages [23], the same sort optimized to use bulk transfers [90], two radix sorts similarly optimized for small and bulk transfers, a connected components algorithm [57], and a conjugate gradient solver. The matrix multiply and the sample sorts have been instrumented to account for time spent in local computation phases and in communication phases separately such that the time spent in each can be related to the processor and network performance of the machines. The execution times for runs on eight processors are shown in Figure 2.5; the times are normalized to the total execution time on the CM-5 for ease of comparison. The matrix multiply uses matrices of 4 by 4 blocks with 128 by 128 double floats each. The main loop multiplies two blocks while it prefetches the two blocks needed in the next iteration. The results show clearly the CPU and network bandwidth disadvantages of the CM-5. The sample sort sorts an array of 4 million 32-bit integers with arbitrary distribution. The algorithm first samples the keys, then permutes all keys, and finally sorts the local keys on each processor. The version optimized for small messages packs two values per message during the permutation phase while the one optimized for bulk transfers presorts the local values such that each processor sends exactly one message to every other processor. The performance again shows the CPU disadvantage of the CM-5 and

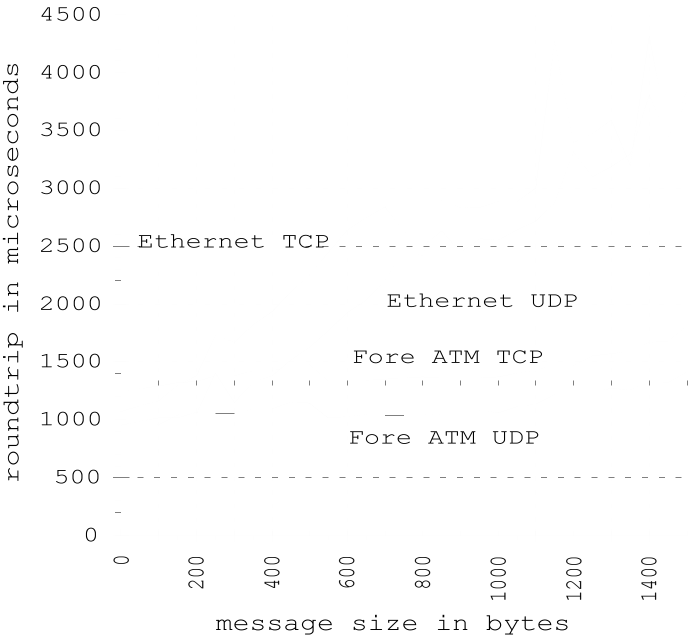


Figure 2.6. TCP and UDP round-trip latencies over ATM and Ethernet as a function of message size

in the small message version that machine's per-message overhead advantage. The ATM cluster and the Meiko come out roughly equal with a slight CPU edge for the ATM cluster and a slight network bandwidth edge for the Meiko. The bulk message version improves the Meiko and ATM cluster performance dramatically with respect to the CM-5 which has a lower bulk-transfer bandwidth. The performance of the radix sort and the connected components benchmarks further demonstrate that the U-Net ATM cluster of workstations is roughly equivalent to the Meiko CS-2 and performs worse than the CM-5 in applications using small messages (such as the small message radix sort and connected components) but better in ones optimized for bulk transfers.

2.7 TCP/IP and UDP/IP protocols

The success of new abstractions often depends on the level to which they are able to support legacy systems. In modern distributed systems the IP protocol suite plays a central role, its availability on many platforms provides a portable base for large classes of applications. Benchmarks are available to test the various TCP/IP and

UDP/IP implementations, with a focus on bandwidth and latency as a function of application message size.

Unfortunately the performance of kernelized UDP and TCP protocols in SunOS combined with the vendor supplied ATM driver software has been disappointing: the maximum bandwidth with UDP is only achieved by using very large transfer sizes (larger than 8Kbytes), while TCP will not offer more than 55% of the maximum achievable bandwidth. The observed round-trip latency, however, is even worse: for small messages the latency of both UDP and TCP messages is larger using ATM than going over Ethernet: it simply does not reflect the increased network performance. Figure 2.6 shows the latency of the Fore-ATM based protocols compared to those over Ethernet.

TCP and UDP modules have been implemented for U-Net using the base-level U-Net functionality. The low overhead in U-Net protocol processing and the resulting low-latency form the basis for TCP and UDP performance that is close to the raw U-Net performance limits presented earlier.

2.7.1 A proof-of-concept implementation

The TCP and UDP over U-Net implementation effort has two goals: first to show that the architecture is able to support the implementation of traditional protocols and second to create a test environment in which traditional benchmarks can be used to put U-Net and kernelized protocol processing into perspective.

By basing the U-Net TCP & UDP implementation on existing software full protocol functionality [14] and interoperability is maintained. A number of modules that were not in the critical performance path were not ported to U-Net, namely the ARP and ICMP modules.

At this point the secure U-Net multiplexor does not have support for the sharing of a single VCI among multiple channels, making it impossible to implement the standard IP-over-ATM transport mechanism, which requires a single VCI to carry all IP traffic for all applications [61]. For IP-over-U-Net a single channel is used to carry all IP traffic between two applications, which matches the standard processing as closely as currently possible. This test setup does not use an exclusive U-Net channel per TCP connection, although that would be simple to implement.

Some issues with IP-over-ATM incompatibility are not yet resolved and are related to what is the best way to implement proper ICMP handling when targeting IPv6

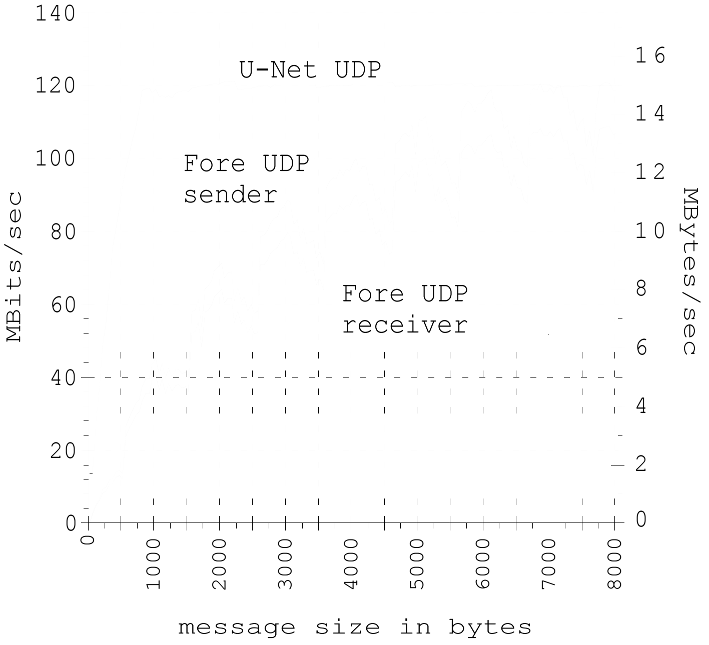


Figure 2.7. UDP bandwidth as a function of message size.

over U-Net. For this an additional level of demultiplexing is foreseen and will be based on the IPv6 [flow-id, source address] tag when packets arrive over the dedicated IP-over-ATM VCI. Packets for which the tag does not resolve to a local U-Net destination will be transferred to the kernel communication endpoint for generalized processing and possibly triggering ICMP handling. This will yield an implementation that is fully interoperable with other IP-over- ATM implementations and will cover both local and wide-area communication.

2.7.2 The protocol execution environment

The TCP/IP suite of protocols is frequently considered to be ill- suited for use over high-speed networks such as ATM. However, experience has shown that the core of the problems with TCP/IP performance usually lie in the particular *implementations* and their *integration* into the operating system and not with the protocols themselves. This is indeed the case with the Fore driver software which tries to deal with the generic low-performance buffer strategies of the BSD based kernel.

Using U-Net, the protocol developer does not experience a restrictive environment (like the kernel) where the use of generalized buffer and timer mechanisms is mandatory and properties of network and application can not be incorporated in the protocol operation. U-Net gives the developer the freedom to design protocols and protocol support software such as timer and buffer mechanisms that are optimized for the particular application and the network technology used. This yields a toolbox approach to protocol and application construction where designers can select from a variety of protocol implementations.

As a result, U-Net TCP and UDP deliver the low-latency and high bandwidth communication expected of ATM networks without resorting to excessive buffering schemes or the use of large network transfer units, while maintaining interoperability with non-U-Net implementations.

2.7.3 Message handling and staging

One of the limiting factors in the performance of kernel-based protocols is the bounded kernel resources available, which need to be shared between many potential network-active processes. By implementing protocols at user-level, efficient solutions are available for problems which find their origin in the use of the operating system kernel as the single protocol processing unit. Not only does U-Net remove all copy operations from the protocol path but also it allows for the buffering and staging strategies to depend on the resources of the application instead of the scarce kernel network buffers.

An example is the restricted size of the socket receive buffer (max. 52Kbytes in SunOS), which has been a common problem with the BSD kernel communication path: already at Ethernet speeds buffer overrun is the cause of message loss in the case of high bandwidth UDP data streams. By removing this restriction, the resources of the actual recipient, instead of those of the intermediate processing unit, now become the main control factor and this can be tuned to meet application needs and be efficiently incorporated into the end-to-end flow-control mechanisms.

The deficiencies in the BSD kernel buffer (*mbuf*) mechanism have been identified long ago [19] and the use of high-performance networks seem to amplify the impact of this mechanism even more, especially in combination with the Fore driver buffering scheme. Figure 2.7 shows the UDP throughput with the saw-tooth behavior that is caused by the buffer allocation scheme where first large 1Kbyte buffers are filled with data and the remainder, if less than 512 bytes, is copied into small mbufs of 112 bytes each. This allocation method has a strong degrading effect on the performance of the protocols because the smaller mbufs do not have a reference count mechanism unlike the large cluster buffers.

Although an alternative kernel buffering mechanism would significantly improve the message handling in the kernel and certainly remove the saw-tooth behavior seen in Figure 2.7, it is questionable if it will contribute as significantly to latency reduction as, for example, removing kernel-application copies entirely [54] .

Base-level U-Net provides a scatter-gather message mechanism to support efficient construction of network buffers. The data blocks are allocated within the receive and transmit communication segments and a simple reference count mechanism added by the TCP and UDP support software allows them to be shared by several messages without the need for copy operations.

2.7.4 Application controlled flow-control and feedback

One of the major advantages of integrating the communication subsystem into the application is that the application can be made aware of the state of the communication system and thus can take application specific actions to adapt itself to changing circumstances. Kernel based communication systems often have no other facility than to block or deny a service to an application, without being able to communicate any additional information.

At the sending side, for example, feedback can be provided to the application about the state of the transmission queues and it is simple to establish a backpressure mechanism when these queues reach a high-water mark. Among other things, this overcomes problems with the current SunOS implementation, which will drop random packets from the device transmit queue if there is overload without notifying the sending application.

Other protocol specific information such as retransmission counters, round trip timers, and buffer allocation statistics are all readily available to the application and can be used to adapt communication strategies to the status of the network. The receive window under U-Net/TCP, for example, is a direct reflection of the buffer space at the application and not at the intermediate processing unit, allowing for a close match between application level flow control and the receive-window updates.

2.7.5 IP

The main functionality of the IP protocol is to handle the communication path and to adapt messages to the specifics of the underlying network. On the receiving side IP-over-U-Net is liberal in the messages that it accepts, and it implements most of the

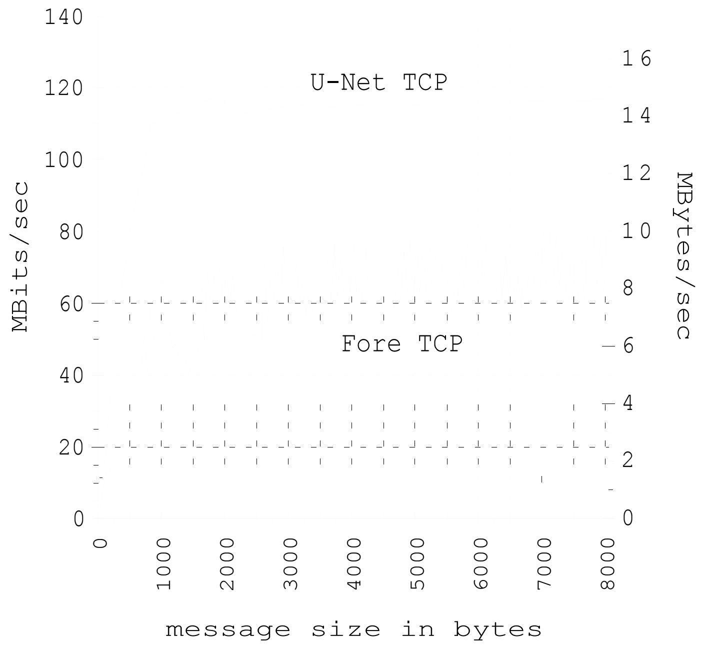


Figure 2.8. TCP bandwidth as a function of data generated by the application

IP functionality, except for the forwarding of messages and the interfacing to ICMP. A transport protocol is selected and the U-Net demultiplex information is passed on to the transport module to possibly assist in destination selection.

On the sending side the functionality of the IP protocol is reduced to mapping messages into U-Net communication channels. Because of this reduced functionality, this side of the protocol is collapsed into the transport protocols for efficient processing.

IP over U-Net exports an MTU of 9Kbytes and does not support fragmentation on the sending side as this is known to be a potential source for wasting bandwidth and triggering packet retransmissions [55]. TCP provides its own fragmentation mechanism and because of the tight coupling of application and protocol module it is relatively simple for the application to assist UDP in achieving the same functionality.

2.7.6 UDP

The core functionality of UDP is twofold: an additional layer of demultiplexing over IP based on port identifiers and some protection against corruption by adding a 16 bit checksum on the data and header parts of the message. In the U-Net implementation the demultiplexing is simplified by using the source endpoint information passed-

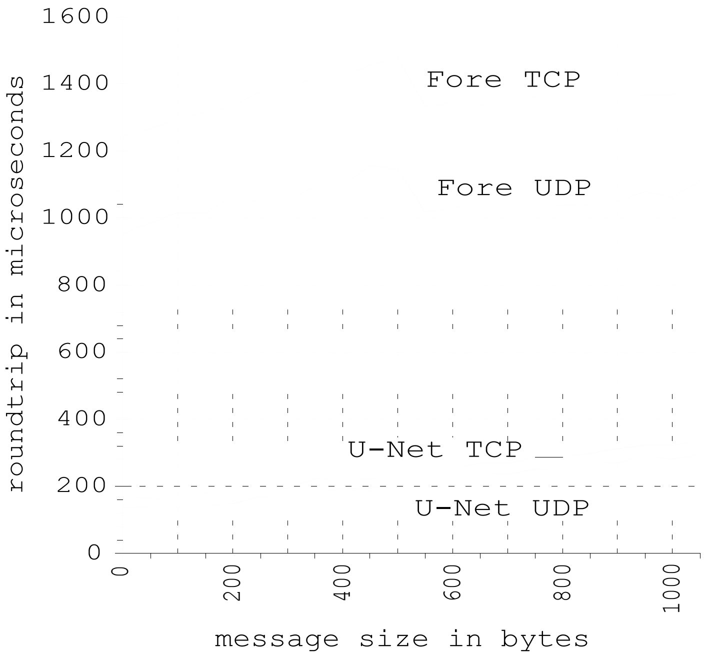


Figure 2.9. UDP and TCP round-trip latencies as a function of message size

on by U-Net. A simple caching scheme of the control data structures (pcb) per incoming channel allows for significant processing speedups, as described by [74]. The checksum adds a processing overhead of 1µs per 100 bytes on a SPARCStation 20 and can be combined with the copy operation that retrieves the data from the communication segment. It can also be switched off by applications that use data protection at a higher level or are satisfied with the 32-bit CRC at the U-Net AAL5 level.

The performance of U-Net UDP is compared to the kernel based UDP in Figures 2.7 and 2.9. The first shows the achieved bandwidth while the latter plots the end-to-end round-trip latency as a function of message size. For the kernel UDP the bandwidth is measured as perceived at the sender and as actually received: the losses can all be attributed to kernel buffering problems at both sending and receiving hosts. With the same experimental set-up, U-Net UDP does not experience any losses and only the receive bandwidth is shown.

2.7.7 TCP

TCP adds two properties that make it an attractive protocol to use in a number of settings: reliability and flow control. Reliability is achieved through a simple acknowledgment scheme and flow control through the use of advertised receive windows.

The performance of TCP does not depend as much on the rate with which the data can be pushed out on the network as on the product of bandwidth and round-trip time, which indicates the amount of buffer space needed to maintain a steady reliable high speed flow. The window size indicates how many bytes the module can send before it has to wait for acknowledgments and window updates from the receiver. If the updates can be returned to the sender in a very timely manner only a relatively small window is needed to achieve the maximum bandwidth. Figure 2.8 shows that in most cases U-Net TCP achieves a 14-15 Mbytes/sec bandwidth using an 8Kbyte window, while even with a 64K window the kernel TCP/ATM combination will not achieve more than 9-10 Mbytes/sec. The round-trip latency performance of both kernel and U-Net TCP implementations is shown in Figure 2.9 and highlights the fast U-Net TCP round-trip, which permits the use of a small window.

2.7.8 TCP tuning

TCP over high-speed networks has been studied extensively, especially over wide-area networks [50] and several changes and extensions have been proposed to make TCP function correctly in settings where a relatively high delay can be expected. These changes need to be incorporated into the U-Net TCP implementation if it is to function across wide-area links where the high latencies no longer permit the use of small windows.

It has been argued lately that the same changes are also needed for the local area case in order to address the deficiencies that occur because of the high latency of the ATM kernel software. U-Net TCP shows that acceptable performance can be achieved in LAN and MAN settings without any modifications to the general algorithms, without the use of large sequence numbers, and without extensive buffer reservations.

Tuning some of the TCP transmission control variables is not without risk when running over ATM [86] and should be done with extreme caution. The low latency of U-Net allows for very conservative settings, therefore minimizing the risk while still achieving maximum performance.

An important tuning factor is the size of the segments that are transmitted: using larger segments it is more likely that the maximum bandwidth can be achieved in cases where low latency is not available. Romanov & Floyd’s work [86] however

Protocol	Round-trip latency	Bandwidth 4K packets
Raw AAL5	65 µs	120 Mbits/s
Active Messages	71 µs	118 Mbits/s
UDP	138 µs	120 Mbits/s
TCP	157 µs	115 Mbits/s
SPlit-C store	72 µs	118 Mbits/s

Table 2.3. U-Net latency and bandwidth summary.

has shown that TCP can perform poorly over ATM if the segment size is large, due to the fact that the underlying cell reassembly mechanism causes the entire segment to be discarded if a single ATM cell is dropped. A number of solutions are available, but none provide a mandate to use large segment sizes. The standard configuration for U-Net TCP uses 2048 byte segments, which is sufficient to achieve the bandwidth shown in Figure 2.8.

Another popular approach to compensate for high latencies is to grow the window size. This allows a large amount of data to be outstanding before acknowledgments are expected back in the hope to keep the communication pipe filled. Unfortunately, increasing the window has its drawbacks. First of all, the large amount of data must be buffered to be available for retransmission. Furthermore, there is a risk of triggering the standard TCP congestion control mechanism whenever there are two or more segments dropped within a single window. Tuning the window size to a large value will increase the chance of this situation occurring, resulting in a drain of the communication pipe and a subsequent slow-start. It seems unavoidable to run these risks, even in a LAN setting, when the protocol execution environment is not able to guarantee low-latency communication.

A final tuning issue that needed to be addressed within U-Net TCP is the bad ratio between the granularity of the protocol timers and the round-trip time estimates. The retransmission timer in TCP is set as a function of the estimated round trip time, which is in the range from 60 to 700 microseconds, but the BSD kernel protocol timer (*pr_slow_timeout*) has a granularity of 500 milliseconds. When a TCP packet is discarded because of cell loss or dropped due to congestion, the retransmit timer is set to a relatively large value compared to the actual round-trip time. To ensure timely reaction to possible packet loss U-Net TCP uses a 1-millisecond timer granularity, which is constrained by the reliability of the user- level timing mechanisms.

The BSD implementation uses another timer (*pr_fast_timeout*) for the transmission of a delayed acknowledgment in the case that no send data is available for piggybacking and that a potential transmission deadlock needs to be resolved. This timer is used to delay the acknowledgment of every second packet for up to 200 ms. In U-Net TCP it was possible to disable the delay mechanism and thereby achieve more reliable performance. Disabling this bandwidth conserving strategy is justified by the low cost of an active acknowledgment, which consists of only a 40-byte TCP/IP header and thus can be handled efficiently by single-cell U-Net reception. As a result, the available send window is updated in the timeliest manner possible laying the foundation for maximal bandwidth exploitation. In wide-area settings, however, the bandwidth conservation may play a more important role and thus the delayed acknowledgment scheme may have to be enabled for those situations.

2.8 Summary

The two main objectives of U-Net—to provide efficient low- latency communication and to offer a high degree of flexibility—have been accomplished. The processing overhead on messages has been minimized so that the latency experienced by the application is dominated by the actual message transmission time. Table 2.3 summarizes the various U-Net latency and bandwidth measurements. U-Net presents a simple network interface architecture which simultaneously supports traditional inter-networking protocols as well as novel communication abstractions like Active Messages.

Using U-Net the round-trip latency for messages smaller than 40 bytes is about 65 µsec. This compares favorably to other research results: the *application device channels* (U. of Arizona) achieve 150 µsec latency for single byte messages and 16 byte messages in the HP Jetstream environment have latencies starting at 300 µsec. Both research efforts however use dedicated hardware capable of over 600 Mbits/sec compared to the 140 Mbits/sec standard hardware used for U-Net.

Although the main goal of the U-Net architecture was to remove the processing overhead to achieve low-latency, a secondary goal, namely the delivery of maximum network bandwidth, even with small messages, has also been achieved. With message sizes as small as 800 bytes the network is saturated, while at smaller sizes the dominant bottleneck is the i960 processor on the network interface.

U-Net also demonstrates that removing the kernel from the communication path can offer new flexibility in addition to high performance. The TCP and UDP protocols implemented using U-Net achieve latencies and throughput close to the raw maximum and Active Messages round-trip times are only a few microseconds over the absolute minimum.

The final comparison of the 8-node ATM cluster with the Meiko CS-2 and TMC CM-5 supercomputers using a small set of Split-C benchmarks demonstrates that with the right communication substrate networks of workstations can indeed rival these specially- designed machines. This encouraging result should, however, not obscure the fact that significant additional system resources, such as parallel process schedulers and parallel file systems, still need to be developed before the cluster of workstations can be viewed as a unified resource.

Chapter 3

Evolution of the Virtual Interface Architecture

The introduction of the VIA standard for cluster or system-area networks has opened the market for commercial user-level network interfaces. The authors examine how design decisions in prototype interfaces have helped shape this industry standard

3.1 Introduction

To provide a faster path between applications and the network, most researchers have advocated removing the operating system kernel and its centralized networking stack from the critical path and creating a *user-level network interface*. With these interfaces, designers can tailor the communication layers each process uses to the demands of that process. Consequently, applications can send and receive network packets without operating system intervention, which greatly decreases communication latency and increases network throughput.

Unfortunately, the diversity of approaches and lack of consensus has stalled progress in refining research results into products—a prerequisite to the widespread adoption of these interfaces. In 1997, Intel, Microsoft, and Compaq introduced the Virtual Interface Architecture [33], a proposed standard for cluster or system-area networks. Products based on the VIA have already surfaced, notably GigaNet's GNN1000 network interface (http://www.giganet.com). As more products appear, research into application-level issues can proceed and the technology of user-level network interfaces should mature. Several prototypes—among them U-Net [37] which is described in chapter 2 — have heavily influenced the VIA. In this chapter, we describe the architectural issues and design trade-offs at the core of these prototype designs, including

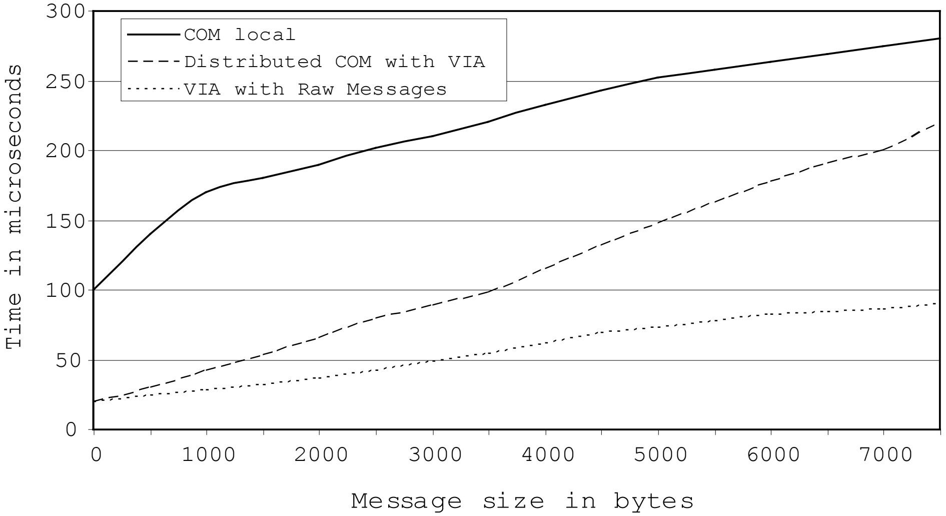


Figure 3.1. Round-trip times for a local remote procedure call (LRPC) under Windows NT (COM local) versus a remote procedure call over the Virtual Interface Architecture using GigaNet's GNN1000 interface (Distributed COM with VIA). By bypassing the kernel the remote call is actually faster than the local one. The round-trip latency of raw messages over the VIA (VIA with raw messages) shows that there is room for improvement in the DCOM protocol.

- How to provide applications with direct access to the network interface hardware, yet retain sufficient protection to ensure that applications can't interfere with each other.
- How to design an efficient, yet versatile programming interface. Applications must be able to access the network interface and still control buffering, scheduling, and addressing. The programming interface, on the other hand, must accommodate a wide variety of hardware implementations.
- How to manage resources, in particular memory. Applications must consider the costs of DMA transfers that map a virtual address to a physical one. At the same time, implementation-specific details must be hidden from the application, and the operating system must ultimately control all resource allocation.
- How to manage fair access to the network without a kernel path which, in traditional protocol stacks, acts as a central point of control and scheduling.

3.2 Performance factors

Network performance is traditionally described by the bandwidth achieved when an infinite stream is transmitted. However, an increasing number of applications are more sensitive to the network's round-trip time, or *communication latency*, and the bandwidth when sending many small messages.

3.2.1 Low communication latency

Communication latency is due mainly to processing overhead—the time the processor spends handling messages at the sending and receiving ends. This may include managing the buffers, copying the message, computing checksums, handling flow control and interrupts, and controlling network interfaces. As Figure 3.1 shows, the round-trip times for a remote procedure call with a user-level network interface can actually be lower than for a local RPC under WindowsNT. The remote call uses Microsoft's distributed component object model (DCOM) with the VIA on the GNN1000 interface. The local call uses Microsoft's component object model (COM). The figure also shows the round-trip latency of raw messages over the VIA, which we used as a baseline in estimating the DCOM protocol overhead.

Low communication latency is key to using clusters in enterprise computing, where systems must be highly available and scalable. Cluster management and cluster-aware server applications rely on multiround protocols to reach agreement on the system's state when there are potential node and process failures. These protocols involve multiple participants—all of which must respond in each round. This makes the protocols extremely sensitive to any latency. Cluster applications that require fault tolerance (for example through a primary/backup scheme or through active replication) use extensive intracluster communication to synchronize replicated information internally. These cluster applications can be scaled up only if the intracluster communication is many times faster than the time in which the systems are expected to respond to their clients. Recent experiments with Microsoft's Cluster Server have shown that without low-latency intracluster communication, the scalability is limited to eight nodes [101].

On the parallel computing front, many researchers use networks of workstations to provide the resources for computationally intensive parallel applications. However, these networks are difficult to program, and the communication costs across LANs must decrease by more than an order of magnitude to address this problem.

3.2.2 High bandwidth for small messages

The demand for high bandwidth when sending many small messages (typically less than 1 Kbyte each) is increasing for the same reasons industry needs low communication latency. Web servers, for example, often receive and send many small messages to many clients. By reducing the per-message overhead, user-level network interfaces attempt to provide full network bandwidth for the smallest messages possible. Reducing the message size at which full bandwidth can be achieved may also benefit data-stream protocols like TCP, whose buffer requirements are directly proportional to the communication latency. The TCP window size, for example, is the product of the network bandwidth and the round-trip time. One way to have maximum bandwidth at minimum cost is to achieve low latency in local area networks, which will keep buffer consumption within reason.

3.2.3 Flexible communication protocols

The traditionally strict boundary between applications and protocols has made it harder to develop more efficient networked applications. Two issues are central: integrating the application and protocol buffer management, and optimizing the protocol control path. In traditional systems, the lack of support for integrated buffer management accounts for a significant part of the processing overhead. This affects application-specific reliability protocols —RPCs, in particular —which must keep data for retransmission. Without shared buffer management, for example, designers cannot use reference count mechanisms to lower the overhead from copying and from transferring buffers between the application and the kernel.

There are several ways to build flexible, optimal communication protocols. Designers can use experimental or highly optimized versions of traditional protocols when integrating the code path of generic and application-specific protocols. Advanced protocol design techniques include *application-level framing*, in which the protocol buffering is fully integrated with application-specific processing, and *integrated-layer-processing*, in which many protocol layers are collapsed into highly efficient, monolithic code paths.

Designers can also use a compiler to compile all protocols together instead of just a small subset of application-specific protocols. Finally, new techniques, such as fast-path generation in the Ensemble communication system [80], use formal verification technology to automatically generate optimized protocol stacks. These techniques

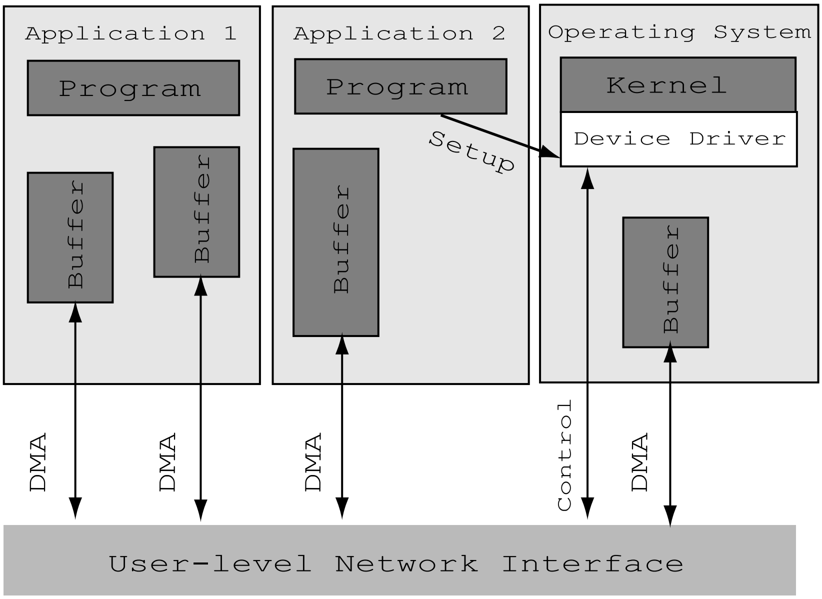


Figure 3.2. System memory with two applications accessing the network using a user-level network interface. A conventional device driver in the operating system controls the interface hardware. The device driver also sets up direct access to the interface for applications(setup). The applications can then allocate buffers in their address space and have the interface transfer message data into and out of these buffers directly using direct memory access(DMA).

rely on control over the complete protocol stack, including the lowest layers, and require the stack to run in a single address space.

3.3 Milestones in interface design

Message-based user-level network interfaces let applications exchange data by sending and receiving explicit messages, similar to traditional multi-computer message-passing interfaces, such as MPI, Intel's NX, and Thinking Machines' CMMD. All the user-level network interfaces we describe let multiple users on a host access the network simultaneously. To provide protection, they separate the communication setup from data transfer. During setup, the operating system is involved and performs protection checks to ensure that applications cannot interfere with each other. During data transfer, the interface bypasses the operating system and performs simple checks to enforce protection.

Figure 3.2 shows system memory with two applications accessing the network through a user-level network interface. A device driver in the operating system controls the interface hardware in a traditional manner and manages the application's access to it.

Applications allocate message buffers in their address space and call on the device driver to set up their access to the network interface. Once set up, they can initiate transmission and reception and the interface can transfer data to and from the application buffers directly using direct memory access.

User-level network interface designs vary in the interface between the application and the network— how the application specifies the location of messages to be sent, where free buffers for reception get allocated, and how the interface notifies the application that a message has arrived. Some network interfaces, such as Active Messages or Fast Messages, provide send and receive operations as function calls into a user-level library loaded into each process. Others, such as U-Net and VIA, expose per-process queues that the application manipulates directly and that the interface hardware services.

3.3.1 Parallel computing roots

Message-based user-level network interfaces have their roots in traditional multicomputer message-passing models. In these models, the sender specifies the data's source memory address and the destination processor node, and the receiver explicitly transfers an incoming message to a destination memory region. Because of the semantics of these send and receive operations, the user-level network interface library must either buffer the messages (messages get transferred via the library's intermediate buffer) or perform an expensive round-trip handshake between the sender and receiver on every message. In both cases, the overhead is high. *Active Messages* [35] were created to address this overhead. Designs based on this notion use a simple communication primitive to efficiently implement a variety of higher level communication operations. The main idea is to place the address of a dedicated handler into each message and have the network interface invoke the handler as soon as the interface receives that message. Naming the handler in the message promotes very fast dispatch; running custom code lets the data in each message be integrated into the computation efficiently.

Thus, Active Message implementations did not have to provide message buffering and could rely on handlers to continually pull messages out of the network. Active

	Myrinet	U-Net	VMMC-2	AM-II	FM	VIA
latency (µsec)	250	13	11	21	11	60
bandwidth (Mb/sec)	15	95	97	31	78	90

Table 3.1. Round-trip performance of standard sockets and user-level network interfaces on Myricom's Myrinet network.

Message implementations also did not need to provide flow control or retransmit messages because the networks in the parallel machines already implemented these mechanisms.

Although Active Messages performed well on the first generation of commercial massively parallel machines, the *immediate* dispatch on arrival became more and more difficult to implement efficiently on processors with deepening pipelines. Some implementations experimented with running handlers in interrupts, but the increasing interrupt costs caused most of them to rely on implicit polling at the end of sends and on explicitly inserted polls. Thus, the overhead problem resurfaced, since polling introduces latency before message arrival is detected and incurs overhead even when no messages arrive.

Illinois *Fast Messages* [73] addressed the immediacy problem by replacing the handler dispatch with buffering and an explicit poll operation. With buffering, Fast Messages can delay running the handlers without backing up the network, and applications can reduce the frequency of polling, which in turn reduces overhead. The send operation specifies the destination node, handler, and data location. The Fast Message transmits the message and buffers it at the receiving end. When the recipient calls are extracted, the Fast Message implementation runs the handlers of all pending messages.

By moving the buffering back into the message layer, the layer can optimize buffering according to the target machine and incorporate the flow control needed to avoid deadlocks and provide reliable communication over unreliable networks. Letting the message layer buffer small messages proved to be very effective—it was also incorporated into the U.C. Berkeley Active Messages II (AM-II) implementations—as long as messages could be transferred un-buffered to their final destination.

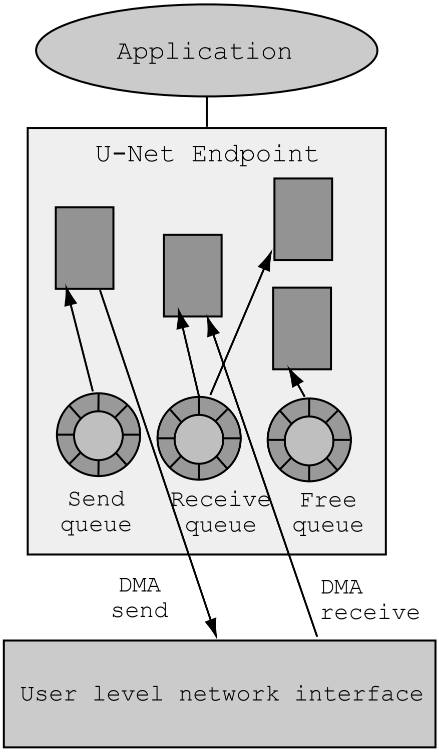


Figure 3.3. U-Net architecture. Each application accesses the network through U-Net endpoints. An endpoint contains send, receive, and free queues that point to buffers the application has allocated. The interface hardware accesses the queues to determine which buffer to transfer messages into, for reception (DMA receive), and out of, for transmission (DMA send).

3.3.2 U-Net

U-Net was the first design to significantly depart from both Active Messages and Fast Messages. Table 3.1 summarizes the performance of U-Net and four subsequent designs as implemented on Myricom's Myrinet network [13]. U-Net provides an interface to the network that is closer to the functionality typically found in LAN interface hardware. It does not allocate any buffers, perform any implicit message buffering, or dispatch any messages. Instead of providing a set of API calls, as in previous interfaces, it consists of a set of in-memory queues that convey messages to and from the network. In essence, Active Messages and Fast Messages define a very thin layer of software that presents a uniform interface to the network hardware. U-Net, on the other hand, specifies the hardware's operation so that the hardware presents a standard interface directly to user-level software.

The U-Net approach offered the hope of a better fit into a cluster environment. Such an environment typically uses standard network technology such as Fast Ethernet or Asynchronous Transfer Mode, and the network must handle not only parallel programs, but also more traditional stream-based communication. Machines within the cluster communicate with outside hosts using the same network.

3.3.3 Architecture

In U-Net, end points serve as an application's handle into the network and contain three circular message queues, as Figure 3.3 shows. The queues hold descriptors for message buffers that are to be sent (send queue), are free to be received (free queue), and have been received (receive queue). To send a message, a process queues a descriptor into the send queue. The descriptor contains pointers to the message buffers, their lengths, and a destination address. The network interface picks up the descriptor, validates the destination address, translates the virtual buffer addresses to physical addresses, and transmits the data using direct memory access (DMA).

When the network interface receives a message, it determines the correct destination end point from the header, removes the needed descriptors from the associated free queue, translates their virtual addresses, transfers the data into the memory using DMA, and queues a descriptor into the end point's receive queue. Applications can detect the arrival of messages by polling the receive queue, by blocking until a message arrives (as in a Unix select system call), or by receiving an asynchronous notification (such as a signal) when the message arrives.

3.3.4 Protection

In U-Net, processes that access the network on a host are protected from one another. U-Net maps the queues of each end point only into the address space of the process that owns the end point, and all addresses are virtual. U-Net also prevents processes from sending messages with arbitrary destination addresses and from receiving messages destined to others. For this reason, processes must set up communication channels before sending or receiving messages. Each channel is associated with an end point and specifies an address template for both outgoing and incoming messages. When a process creates a channel, the operating system validates the templates to enforce system-specific policies on outgoing messages and to ensure that all incoming messages can be assigned unambiguously to a receiving end point.

The exact form of the address template depends on the network substrate. In versions for ATM, the template simply specifies a virtual channel identifier; in versions for Ethernet, a template specifies a more elaborate packet filter.

U-Net does not provide any reliability guarantees beyond that of the underlying network. Thus, in general, messages can be lost or can arrive more than once.

3.3.5 AM-II and VMMC

Two models provide communication primitives inspired by shared memory: the AM-II, a version of Active Messages developed at the University of California at Berkeley [17], and the Virtual Memory Mapped Communication model, developed at Princeton University as part of the Shrimp cluster project [12].

The primitives eliminate the copy that both fast messages and U-Net typically require at the receiving end. Both Fast Message implementations and U-Net receive messages into a buffer and make the buffer available to the application. The application must then copy the data to a final destination if it must persist after the buffer is returned to the free queue.

AM-II provides put and get primitives that let the initiator specify the addresses of the data at the remote end: put transfers a local memory block to a remote address; get fetches a remote block. VMMC provides a primitive essentially identical to put. In all these primitives, no receiver intervention is necessary to move the data to the final location as long as the sender and receiver have previously coordinated the remote memory addresses. AM-II associates an Active Message handler with each put and get VMMC provides a separate notification operation.

VMMC requires that communicating processes pin down all memory used for communication so that it cannot be paged. VMMC-2 [31] lifts this restriction by exposing memory management as a *user-managed translation look-aside buffer*. Before using a memory region for sending or receiving data the application must register the memory with VMMC-2, which enters translations into the UTLB. VMMC-2 also provides a default buffer into which senders can transmit data without first asking the receiver for a buffer address.

3.3.6 Virtual Interface Architecture

The VIA combines the basic operation of U-Net, adds the remote memory transfers of VMMC, and uses VMMC-2's UTLB. Processes open *virtual interfaces* (VI) that

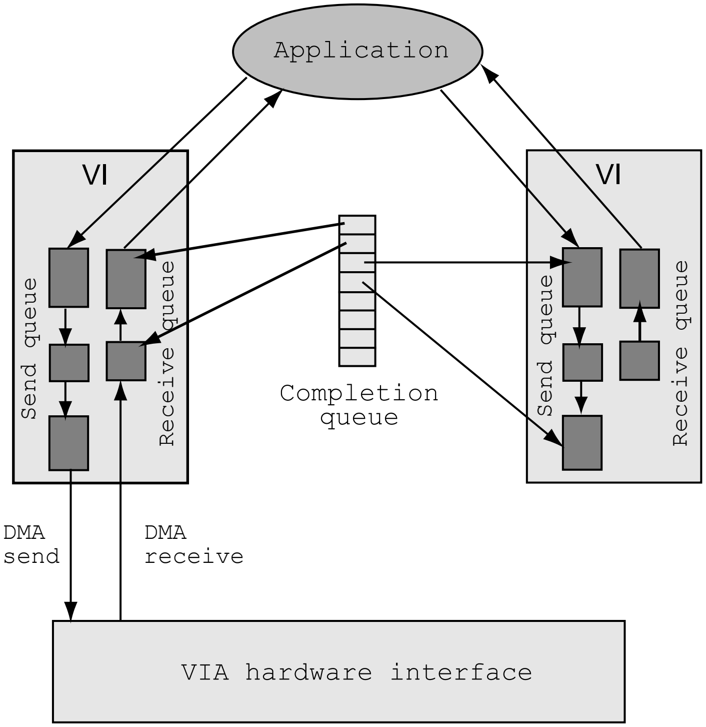


Figure 3.4. Queuing in the Virtual Interface Architecture. Each application can open multiple virtual interfaces (VIs), each with its own send and receive queues. Each VI is associated with a completion queue to which the VIA adds a descriptor for every completed transmission or reception. By sharing one completion queue across many VIs, an application must check only a single queue to pick up all events related to its network connections.

represent handles onto the network, much like U-Net's end points. As Figure 3.4 shows, each VI has two associated queues—send and receive—that are implemented as linked lists of message descriptors. Each descriptor points to one or multiple buffer descriptors. To send a message an application adds a new message descriptor to the end of the send queue. After transmitting the message, the VIA sets a completion bit in the descriptor, and the application eventually takes the descriptor out of the queue when it reaches the queue's head. For reception, the application adds descriptors for free buffers to the end of the receive queue, which VIA fills as messages arrive.

Each VI represents a connection to a single other remote VI. This differs from the U-Net end point, which can aggregate many channels. In the VIA, a process can create one or more *completion queues* and associate each with multiple VIs. The network interface fills entries in the completion queue to point to entries in the send or receive queue that have been fully processed.

The VIA also provides direct transfers between local and remote memory. These *remote* DMA writes and reads are similar to AM-II's puts and gets and VMMC's transfer primitives.

To provide some protection, the VIA lets a process specify which regions of its memory are available for RDMA operations. Memory management in the VIA is very similar to the VMMC-2. A UTLB resides in the network interface. All memory used for communication must be registered with the VIA before it is used, including all queues, descriptors, and buffers. Registering a region returns a handle, and all addresses must be specified using the appropriate region handle and a virtual address.

3.4 Design trade-offs

As the previous description shows, the user-level network interface designs that have shaped the VIA differ from each other in many respects. These differences reflect attempts to optimize against network hardware and programming environments. Trade-offs are most apparent in the choice of queue structure, memory management strategy, and message multiplexing approach.

3.4.1 Queue structure

The queues in U-Net and the VIA expose a structure very similar to most hardware network devices. Exposing the queues instead of providing a procedural interface (as in Fast Messages and Active Messages) means that higher software layers can manage the required descriptors and buffers directly. This, in turn, means that at no point is an application's executing thread required to yield control to complete an operation. Exposing the queues also very naturally splits an operation's initiation from its completion. Pushing a descriptor into the send queue initiates the transmission; the network interface signals completion using the descriptor. The actual transmission or reception can thus occur asynchronously to the ongoing computation.

The queues are designed to accept *scatter-gather* descriptors, which allow applications to construct messages from a collection of noncontiguous buffers. For example, in a sliding window protocol (in which the transmission window moves over a large data buffer), the protocol implementation places its information in a separate buffer. The network interface then concatenates the data and protocol information.

3.4.2 Memory management

Integrating buffer management between the application and the network interface is important in eliminating data copies and reducing allocations and de-allocations. This integration gives rise to additional complexity, however, because the application uses virtual addresses for its message buffers, whereas the network interface DMA engine requires physical memory addresses for its transfers. Thus, a trusted entity must translate a virtual address to a physical one each time a buffer is handed to the network interface. In addition, the operating system must track the memory pages available to the interface for DMA transfers so that the operating system can keep mappings constant.

The VIA's memory management is inspired by VMMC-2's UTLB, which makes the application responsible for managing virtual-to-physical address translations. The application must request a mapping for a memory region from the operating system before using any buffers in that region. The advantage is that the operating system can install all mappings in the network interface, which can then translate all virtual addresses using a simple lookup. The primary drawback is that the application must take care to manage its memory regions judiciously. In simple cases, a fixed set of buffers is allocated, so it is easy to keep the memory region handles in the buffer descriptors. However, if the application transmits data out of internal data structures directly, it may have to request a new translation for every transmission, which is time-consuming.

In U-Net, in contrast, applications can place buffers anywhere in their virtual address space and the network interface does the virtual-to-physical address translation [105]. For this reason, the interface incorporates a TLB that maps <*process ID, virtual address*> pairs to physical page frames and read/write access rights.

The disadvantage of U-Net's approach is that the application may present an address that has no mapping in the interface's TLB. If the address is for a buffer to be sent, the interface must request a translation from the host kernel, which requires a somewhat complex and costly handshake (approximately 20 µs of overhead in Windows NT, for example).

To avoid TLB misses when a message arrives, the interface must pre-translate entries in each free buffer queue. If a message arrives and no pre-translated buffer is available, the interface drops the message. Requesting translation on-the-fly would be very difficult because reception occurs within an interrupt handler, which has only limited access to kernel data structures.

An important benefit of U-Net's memory-allocation scheme is that the host kernel can limit the amount of memory pinned down for the network by limiting the number of valid entries in the interface's TLB. This is somewhat analogous to pre-allocating and pinning a fixed-size buffer pool in kernel-based network stacks. The difference is that, in U-Net, the set of pages in the buffer pool can vary over time.

Shifting control over memory management from the operating system to the application, as the VIA does, has a potentially more serious drawback. If an application requests a mapping, the operating system must grant it or risk an application failure. Once it grants a mapping, the operating system cannot revoke it to shift resources, for example, to a higher priority process. If servers and workstations host multiple applications with simultaneous open network connections, and each application uses, say 200 to 300 Kbytes of buffer space, a significant fraction of the physical memory must be dedicated to network activity.

In U-Net, in contrast, only entries in the interface's TLB must be pinned in memory. The interface can also remove mappings of applications not actively using their network buffers, thereby making the memory eligible for paging and available to applications in general.

3.4.3 Multiplexing and de-multiplexing

In user-level network interfaces, the network interface must de-multiplex arriving messages onto the correct receive queue. Similarly, when sending a message, the interface must enforce protection (by validating addressing information, for example). The complexity of both operations depends heavily on the type of network used.

The VIA's approach to multiplexing is strictly connection oriented. Two virtual interfaces must set up a connection before any data can be transmitted. The specification does not prescribe how the connection must be represented at the network level, so designers could run the VIA over an Ethernet or even over the Internet Protocol by defining some handshake between the end points and by using a special field to differentiate connections. Because of the VIA's connection orientation, applications that use VIA networks cannot interoperate with applications that use non-VIA networks. For example, a server cluster using the VIA internally must use conventional network interfaces to communicate with clients outside the cluster.

U-Net, in contrast, makes the multiplexing and de-multiplexing processes much more flexible. In the Fast Ethernet version, the network interface implements a full packet filter, which is not connection oriented. When an application opens a communication channel, it specifies patterns for parsing the headers of incoming and outgoing packets. The kernel agent validates the patterns to ensure that there will be no conflict with other applications and enters them into the multiplexing data structures. The generic nature of U-Net's packet processing means that communication with a generic protocol stack is possible, as is the sending and receiving of multicast messages.

The filtering must be implemented as a dynamic packet parser to support a variety of network protocols and accommodate the nature of identifying fields, which are not at fixed locations in messages. The packet filter must be more sophisticated than those used previously [4] because there is no backup processing path through the operating system kernel for handling unknown packets. Such a fail-safe path is not an option because the architecture would then force the operating system to jointly manage the receive queues with the interface. The Fast Ethernet version of U-Net sidesteps the issue by using a standard network interface and implementing its functionality in the interrupt handler, where the complexity of the packet filter is tolerable.

3.4.4 Remote memory access

Handling RDMA reads and writes adds considerable complexity to VIA implementations. When receiving a write, the interface must not only determine the correct destination VI but also extract the destination memory address from the message and translate it. The main issue here is how to handle transmission errors correctly: The interface must verify the packet checksum before any data can be stored into memory to ensure that no error affected the destination address. This verification is not trivial because most networks place the checksum at the end of the packet, and the interface must buffer the entire packet before it can start the DMA transfer to main memory. Alternatively, a separate checksum could be included in the header itself.

RDMA reads also present a problem. The network interface may receive read requests at a higher rate than it can service, and it will have to queue the requests for later processing. If the queue fills up, requests must eventually be dropped. To ensure a safe implementation of RDMA reads, the VIA restricts read requests to implementations that provide reliable delivery.

3.5 Conclusions

VIA designers succeeded in picking most of the cherries from extensive research in user-level interfaces and in providing a coherent specification. Users familiar with this research will have little difficulty in adapting to the VIA and will enjoy the benefit of multiple off-the-shelf commercial hardware implementations. Because the VIA's primary target is server area networks, however, its designers did not provide one important cherry from U-Net: The VIA does not provide interoperability with hosts that use conventional network interfaces.

To fully exploit the promise of the VIA and of fast networks in general, significant additional research is needed in how to design low-latency protocols, marshal objects with low overhead, create efficient protocol timers, schedule communication judiciously, and manage network memory. Memory management presents particularly difficult trade-offs. The UTLBs in VMMC-2 and the VIA are easy to implement but take away resources from the general paging pool. The TLB proposed in U-Net implements a close interaction with the operating system and provides fair memory management but is more complex and incurs extra costs in the critical path whenever a miss occurs. Thus, although the VIA user-level network interfaces are now available to everyone, many design issues must yet be solved before application writers can enjoy painless benefits. Hopefully, research will break through some of these issues as users begin to document their experiences in using VIA.

Chapter 4

Tree-Saturation Control in the AC3 Velocity Cluster Interconnect

In a multi-user production cluster there is no control over the intra-cluster communication patterns, which can cause unanticipated hot spots to occur in the cluster interconnect. In a multistage interconnect a common side effect of such a hot-spot is the roll-over of the saturation to other areas in the interconnect that were otherwise not in the direct path of the primary congested element. This chapter investigates the effects of tree saturation in the interconnect of the AC3 Velocity cluster, which is a multistage interconnect constructed out of 40 GigaNet switches. The main congestion control mechanism employed at the GigaNet switches is a direct feedback to the traffic source, allowing for fast control over the source of the congestion, avoiding the spread from the congestion area. The experiments reported are designed to examine the effects of the congestion control in detail.

4.1 Introduction.

An important issue in traffic management of multi-stage interconnects is the handling of tree saturation (caused by hot spot traffic [75]), and the impact that tree saturation can have on unrelated flows. In a production multi-user parallel machine such as the AC3 Velocity cluster, this is particularly important as traffic patterns are not predictable, and hot-spots cannot be avoided through application level structuring.

There are two aspects of the handling of the saturation effects that are of primary importance; first there is the fairness among the flows that travel through a region of the switch fabric that contains a 'hot-spot'; flows that cause the congestion should be reduced equally and fairly to relieve the congested link.

Secondly there are the effects on flows that are not traveling over congested links, but that do cross switches that are part of the tree that is saturated. Foremost of those effects is second order head-of-line blocking, which can occur even if the individual switches are constructed to handle head-of-line blocking gracefully.

This chapter describes the GigaNet multi-stage interconnect of the AC3 Velocity cluster, which is constructed of 40 switching elements organized into a Fat-Tree. A number of techniques are employed in the GigaNet interconnect that control the saturation effects, and that allow the interconnect to gracefully adapt to occurrence of hot-spots. The feedback and flow-control based techniques provide fairness in the scheduling of the competing streams and predictable behavior of unrelated streams that could potentially be impacted by second order effects.

This chapter is organized as follows: in sections 4.2 and 4.3 the cluster and the interconnect are described in detail. Section 4.4 examines the problems that are related to saturation in multistage interconnects, and section 4.5 describes the setup of the experiments to investigate the saturation effects. In section 4.6 the results of the experiments are presented with conclusions and related work following in section 4.7 and 4.8.

4.2 The AC3 Velocity Cluster

AC3 Velocity is composed of 64 Dell PowerEdge servers, each of which has four Intel Pentium III Xeon SMP processors running at 500 Mhz with 2 MB of Level 2 cache per processor. Each Power Edge contains 4 gigabytes RAM and 54 gigabytes of disk space. Microsoft Windows NT 4.0 Server, Enterprise Edition, is the operating system. Each rack holds eight servers. The switch fabric is comprised of 40 Giganet cLAN 8x8 switch elements.

The experimental super computer and cluster facility is based at the Cornell Theory Center: a high performance computing and interdisciplinary research center located at Cornell University. AC3 is the center’s research and IT service consortium for business, higher education, and government agencies interested in the effective planning, implementation, and performance of commodity-based systems, software, and tools.

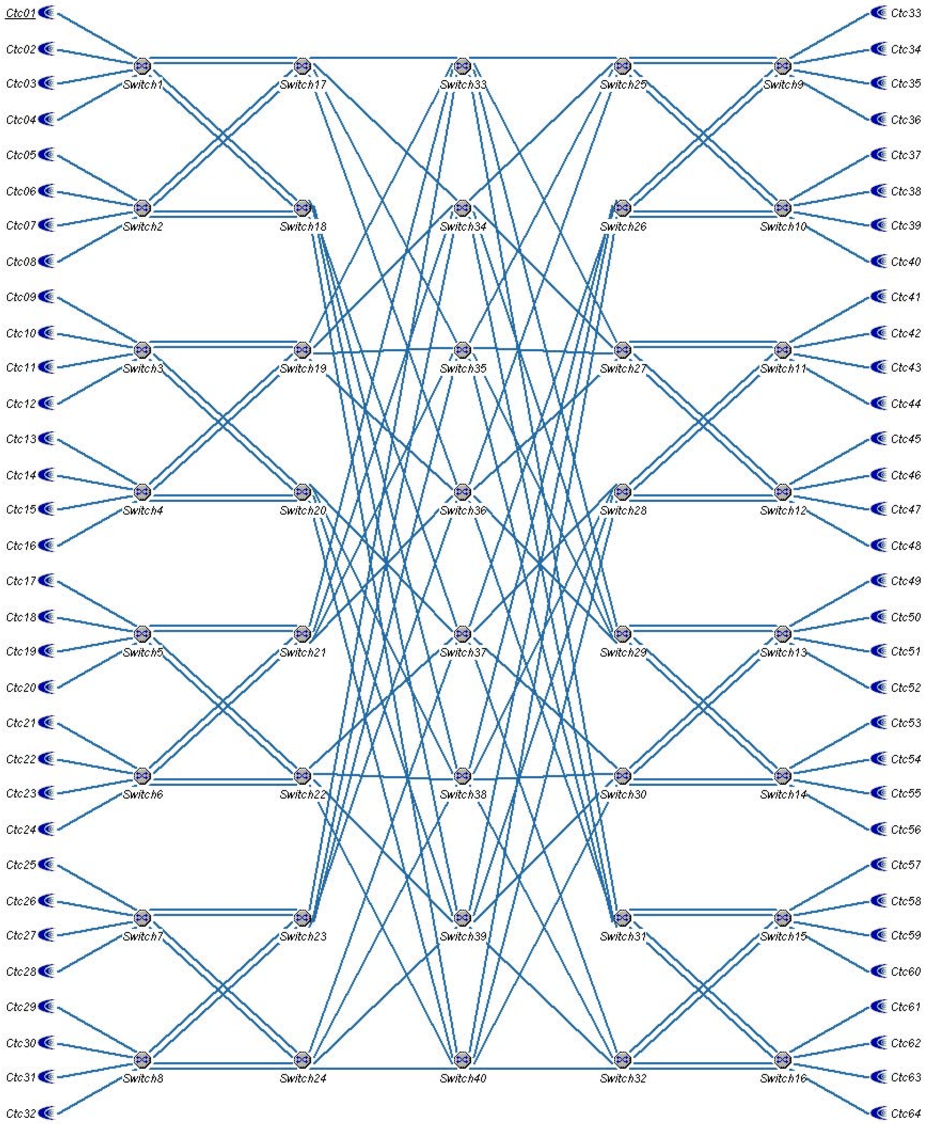


Figure 4.1. Layout of the switches in the AC3 cluster interconnect.

4.3 The GigaNet Interconnect

The interconnect of the AC3 Velocity cluster is a multistage interconnection network constructed out of GigaNet cluster area network (cLan) switching elements and host interfaces.

The host interface provides a hardware implementation of the Virtual Interface (VI) Architecture specification [32], delivering the interconnect's raw latency and bandwidth directly into application processes, while maintaining full security [38].

A cLan switch is designed using a single chip architecture based on GigaNet's proprietary chip for ATM switching. The first generation GigaNet chip is present in the switches that make up the AC3 Velocity interconnect.

Cluster switching fabric switches at 8x1Gb/sec using a non-blocking, shared memory architecture with 16 Gb/sec cross-sectional bandwidth. The switch uses the memory to implement virtual buffer queue architecture, where cells are queued on a per VCI per port basis. The host interface also implements a virtual buffer queue architecture, where cells are queued on a per VCI basis. cLAN switches are shipped in eight port 1U and 32 port 2U configurations. These building blocks can be interconnected in a modular fashion to create various topologies of varying sizes. In the AC3 Velocity Cluster 40 eight port switches are deployed in a fat tree topology as shown in Figure 4.1. Each stage holds 8 switches, which results in that the maximum number of hops between any two nodes in the system is 5.

The use of ATM for transport and routing of messages is transparent to the end host. VI endpoints correspond directly to a VCI, using AAL5 encapsulation for message construction, similar to [37]. Switching is performed on a per VCI basis; and no grouping techniques are used at the switch, as flow control policies are implemented on a per VCI basis.

Congestion is evaluated on a per VCI basis, taking into account VCI, link, and general buffer utilization, as well as general system configuration. If flow control is triggered, the switch will start sending Source Quench indications to VCI source, which will respond immediately by shutting down the source until an unquench indication arrives. The flow control mechanism is implemented in hardware and quenches can be generated at very high frequency. In practice there are always a large numbers of Quench/Unquench indications flowing through the network.

The very high frequency of the flow control indications allows the sources to be bandwidth controlled in a relatively flat manner. It enables the switches that experience potential congestion to schedule the competing streams in a fair manner according to the overall traffic pattern. A second effect of this flow control architecture is that the data sources can be constructed in a simple manner, executing as greedy as possible, relying on the switch flow control indications to perform the traffic shaping.

The clan interconnect is loss-less. A special modification to the clan product allowed flow control to be disabled and replaced with link-level flow control for the purpose of the experiments in this chapter.

4.4 The Problem

Interconnect behavior under a variety of realistic workloads has been studied for a long time and this resulted in improved switch and interconnect designs. One of the problems that has been the hardest to solve is that of congestion management in the face of unpredictable traffic patterns.

Feedback techniques [91] such as multi-lane backpressure [52] have been experimented with and the results are promising. The flow-control techniques in a GigaNet based interconnect are novel in that (1) the feedback is directly to the VCI source and not to the predecessor switch in the path, (2) it does not employ any credit based scheme, and (3) that the flow-control is used to perform traffic shaping in the overall interconnect..

The AC3 velocity cluster provides an excellent opportunity to examine the effectiveness of these techniques given the number of switches in the fabric. There are three particular problem areas that are of interest, when examining congestion control:

1. **Tree-saturation.** When a switch becomes congested will there be saturation roll-over and spread the congestion to other switches in the region?
2. **High-order head-of-line blocking.** Even if individual switches are constructed such that they exhibit no head-of-line blocking when ports become congested, placing them in a multi-stage interconnect may trigger higher-order HOL occurrences because of link and buffer dependencies between switches [51].
3. **Fairness among congested streams.** If a number of streams flow through single congestion point, will the traffic shaping be such that all streams are treated fairly.

To examine these three problem areas a number of experiments have been designed that are described in detail in section 4.6. All experiments were performed with the flow control enabled as well as disabled.

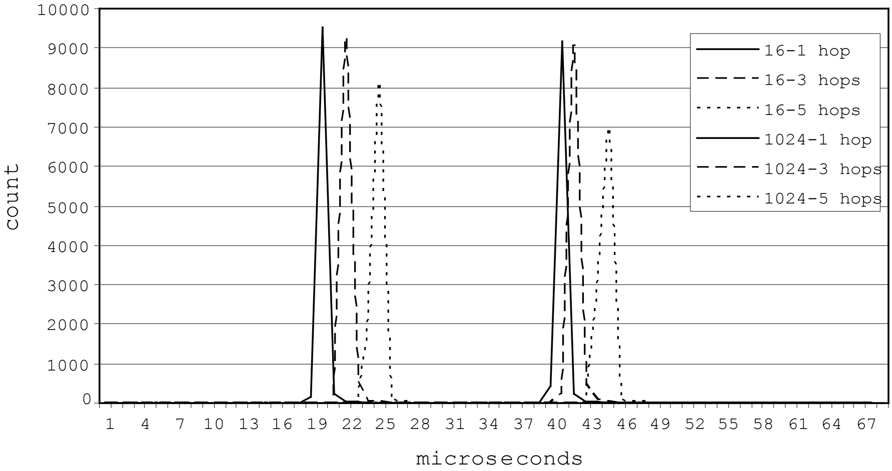


Figure 4.2. latency histogram of 16 and 1024 byte messages per number of hops

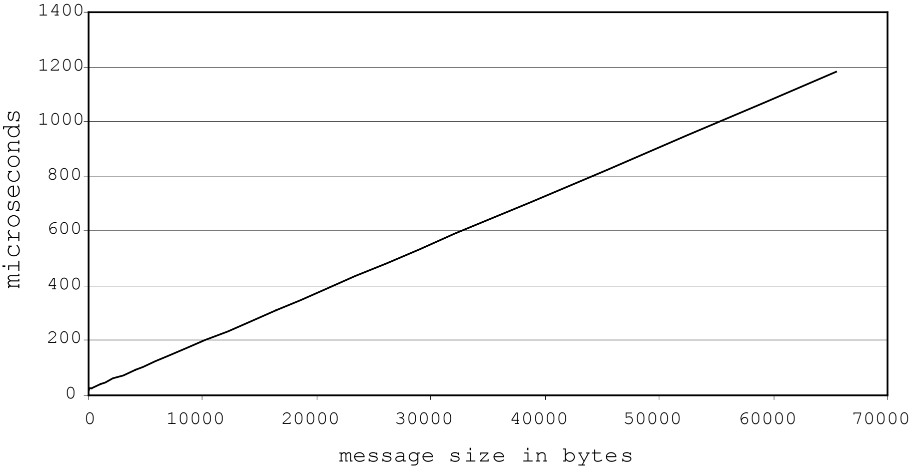


Figure 4.3. Average latency per message size

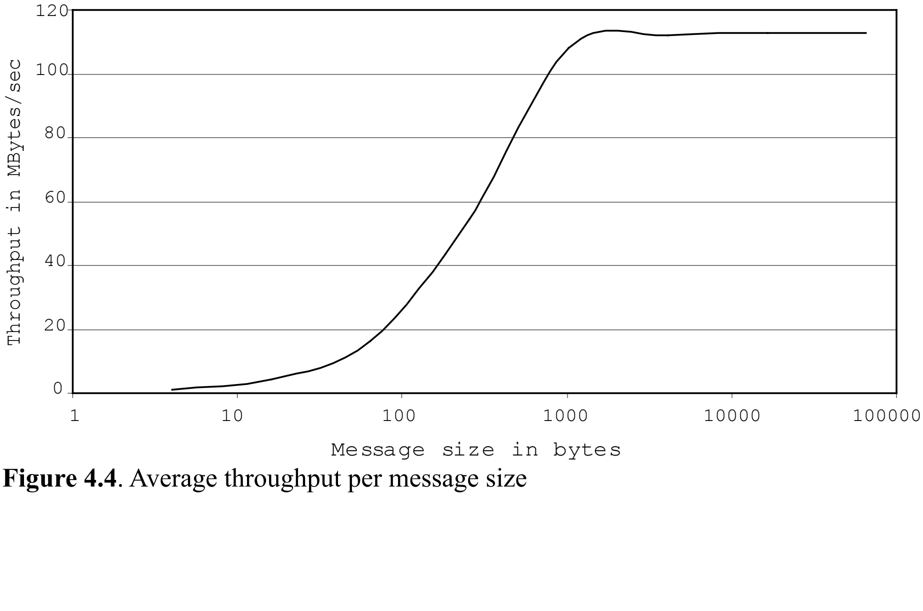


Figure 4.4. Average throughput per message size

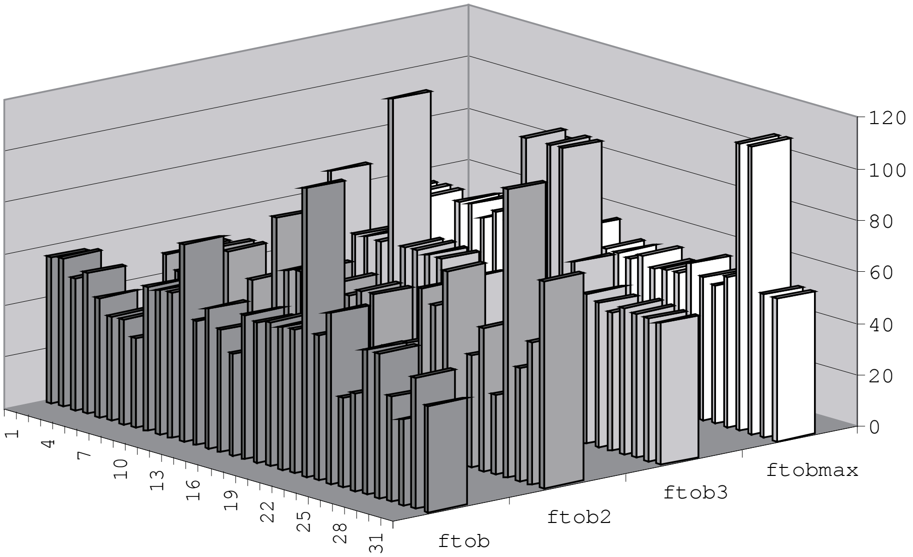


Figure 4.5. Histogram of the throughput in KBytes/sec of the individual streams in 4 different *front-to-back* tests.

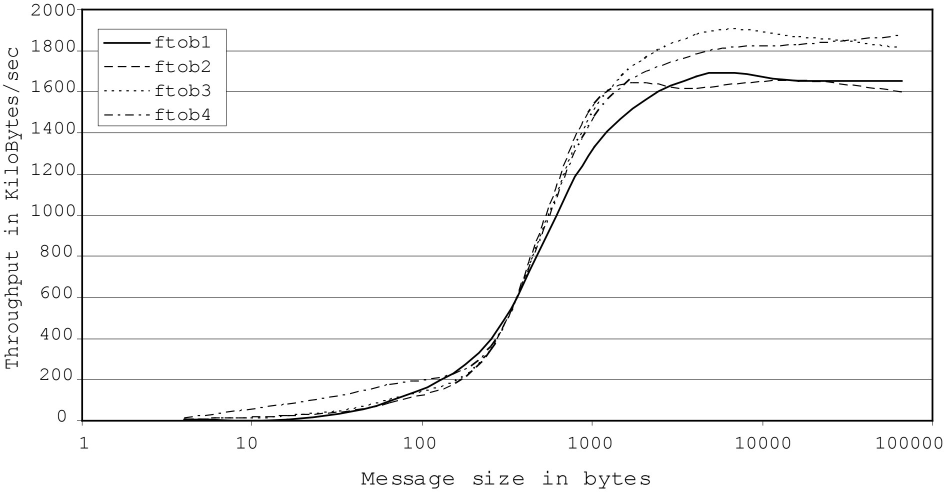


Figure 4.6. Total fabric throughput in KBytes/sec per message size in the different *front-to-back* tests

4.5 Baseline Performance

In this section we briefly touch on the baseline performance of the interconnect. Standard latency and throughput tests were conducted between sets of nodes in the cluster. Bottom line latency is 10 usec, maximum throughput close to 114 Mbytes/sec and the maximum message rate is over 250,000 messages/sec.

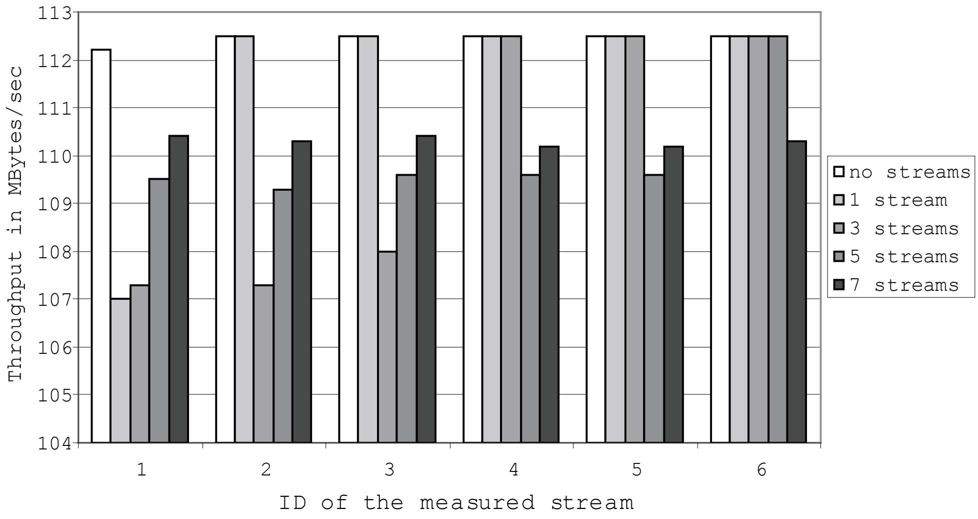


Figure 4.7. The impact of adding congestion streams to the 6 background streams in the *slow-host-congestion-fabric* test.

In figure 4.2 a histogram of latency is shown of 16 and 1024 bytes messages in relation to the number of hops between source and destination. Each additional hop adds 1 usec to the latency. Figure 4.3 shows the average latency with respect to message size. Figure 4.4 shows the bandwidth in relation to message size. For bandwidth measurements of single streams, the number of switches in the stream did not matter.

The maximum message throughput is 266,000 messages/sec, which is limited by the host PCI bus, and which is achieved with messages with 4 bytes payload.

4.6 Tree Saturation Experiments

To investigate the effects of tree saturation we conducted four dedicated tests:

4.6.1 Front-to-back

A test where 32 connections are made between random nodes that all cross the maximum number of stages of the interconnect, triggering hot-spots in the communication. For practical execution of this experiment, all sources are chosen from the nodes 01-32 (the front) while destinations come from nodes 33-64 (the back). Four different connection layouts were tested with a variety of message sizes. Each test was run for 30 seconds and the results were analyzed for variations in inter-

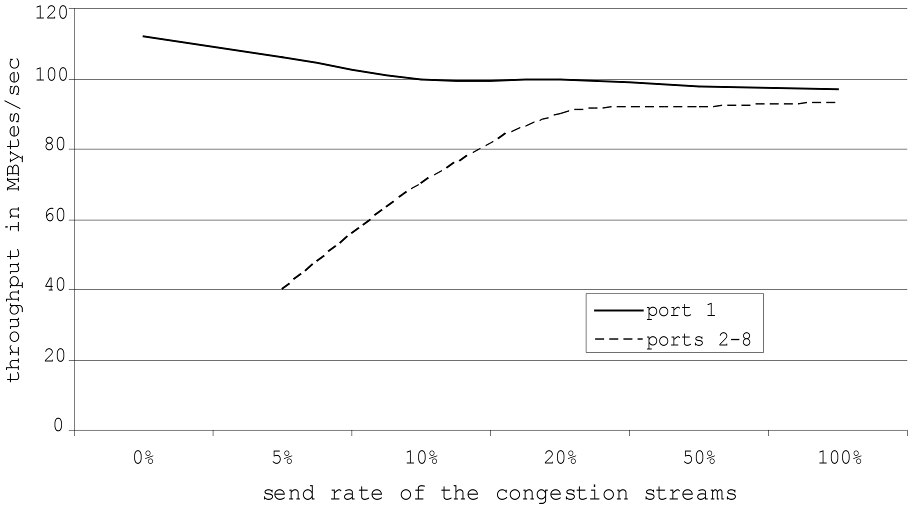


Figure 4.8. Throughput on each output port in the *switch port contention* test.

arrival rates, in bandwidth over time, in comparative bandwidth among the streams and overall throughput of the interconnect in relation to message size

Given the randomness in the connection setup, some hotspots occur within traffic, while there are also some connections that share no links at all. Figure 4.5 shows a histogram of the individual stream throughput measured in the four tests.

Figure 4.6 shows the overall throughput through the interconnect in relation to the message size.

4.6.2 Slow Host Congestion Fabric

This test is used to examine if congestion will spread through the interconnect when a host network interface controller (NIC) becomes congested. In the test up to seven streams will come into node01, each entering a different port on switch01 and exiting on the port connect to node01. The congested NIC will cause switches 01, 17 and 34 to congest, where switch 34 is a 3rd layer switch. Six large streams will also flow through switch 34, each share an input port with the streams directed to node01. In this test the congestion into node01 is varied and its impact on the overall throughput of switch 34 is measured.

Without any traffic directed at node01 each of the stream achieves maximum throughput (112 Mbytes/sec). The streams used to congest the NIC consist of single cell messages (4 bytes payload) and the streams are added stepwise. The first

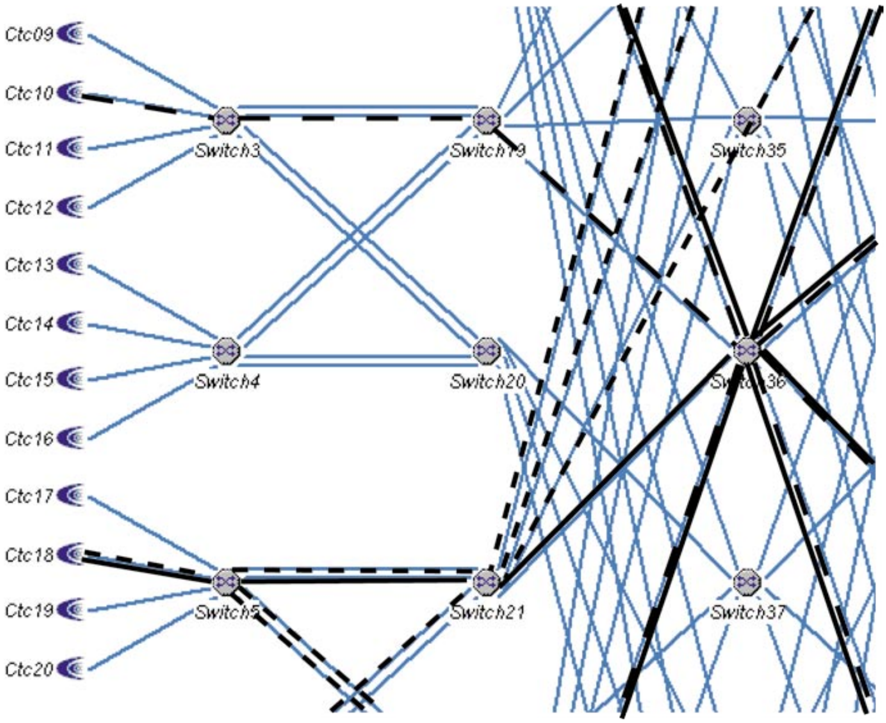


Figure 4.9. Layout of the multi-stage-congestion test. The small dashed strreams are the background traffic, the solid line is the congestion traffic and large dashed line is the side-effect probe

stream reaches the maximum message throughput of 266,000 messages per second, resulting in up to 1.05 Mbytes/sec. This throttles down the background stream that shares the input port with the congestion stream to 107 Mbytes/sec while the other streams remain unaffected.

Adding more streams towards the NIC causes the competing streams to drop towards equal share of the maximum message throughput at the NIC, e.g. with 3 streams each reaches a throughput of 88,000 per second (.35 Mbytes/sec). Each of the background streams that now share an input port with a congested stream, throttle back slightly, but not less then 110 Mbytes/sec.

In the test also one congested stream did not share an input port with a background stream, and this stream did not receive any preferential treatment of the streams that did share input ports.

	Node 18 - I	Node 18 - II	Node 18 total	Node 10
Test 1	115	9	115	0
Test 2	115	0	115	114
Test 3 - 5%	52	31	84	0
Test 3 - 20%	49	42	91	0
Test 3 - 100%	61	53	114	0
Test 4 - 5%	61	53	114	30
Test 4 - 20%	62	54	116	64
Test 5 - 100%	60	54	114	114

Table 4.1. The throughput in MBytes/sec measured at the destination nodes in the multi-stage contention test. Node 18 is divided into teh set coming from switch 5 &12 (I), and from switch 36 (II).

4.6.3 Switch Port Contention

This test exposes whether contention for a single port on a 3rd level switch will affect other traffic flowing through the same switch. In this test there are seven streams entering switch 40 on ports 2-8, while exiting at port 1. Seven other streams are entering the switch at ports 2-8, but exiting the switch through the same set of ports. The contention of port 1 is varied and the effect on the overall throughput is measured.

Starting without the streams that will congest port 1, the seven background streams all achieve continuously the maximum throughput off 112 Mbytes/sec each. When the congestion streams are introduced their rates are varied from 5%-100% of maximum throughput. At 20% each of the competing streams has reached is maximum throughput of 15 Mbytes/sec, resulting in an output throughput of 91 Mbytes/sec. The background streams have been throttled back to 98 Mbytes/sec (see figure 4.8).

Increasing the message rates, on the congested streams has no effect, their individual throughput remains at 15 Mbytes/sec. Each of the seven input links continues to run at maximum throughput with a background stream and a congestion stream coming in on each link. The overall throughput in the switch remains at 780 Mbytes/sec independent of how the input streams are varied. The balance among the streams is close to ideal: the seven background stream throttle back to identical throughput, while the congestion streams each use up 1/7th of the output on port 1.

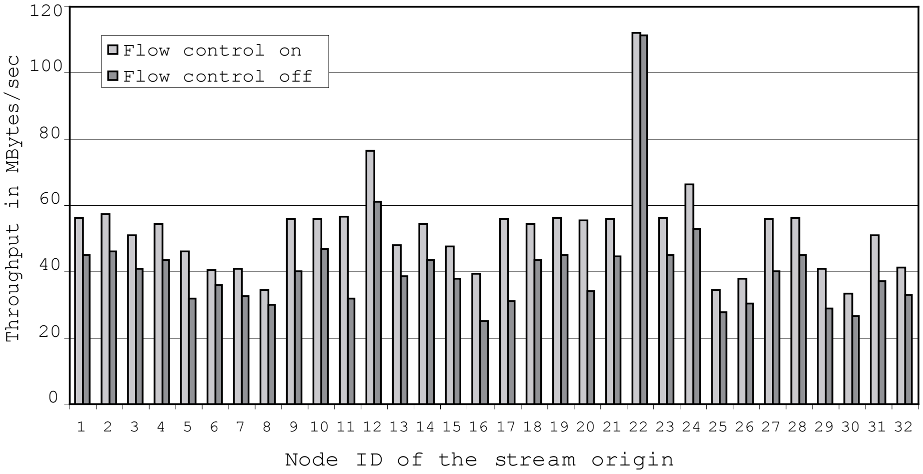


Figure 4.10. The throughput in MBytes/sec achieved in a *front-to-back* test with stream based flow control enabled (light bars) and disabled (dark bars).

4.6.4 Multi-stage Congestion

In this test the effects of congestion in a switch on other switches in the fabric is measured, and the fairness among flows through the congested points is examined. For this test there is a set of six sources that each send to both nodes10 and 18, causing contention to occur in switch 36 at port 2 and 3.

A second set of sources send to node 18, congesting switch 21 and 5. The traffic into nodes 10 and 18 is varied and the effect on the overall throughput is measured as well at the balance between the individual streams (see figure 4.9).

The first test is to only send data from the second set of sources, which enter through switch 21 and 5. Jointly they reach a maximum throughput of the 115Mbytes/sec, which is limited by the single link going into node 18. Each stream receives an equal share of the bandwidth (17 Mbytes/sec).

Secondly the 6 streams flowing through switch 36 to node 10 are added, and the results show that all streams run at maximum throughput.

In the third part of this test the streams to node 10 are stopped and the additional streams for node 18 coming through switch 36 are introduced, in stepwise manner. The total throughput arriving at node 18 drops to 91 Mbytes/sec when the new streams come in at low rates, while higher rates push the throughput up to 114 Mbytes/sec. The throughput is equally divided over the 13 incoming streams.

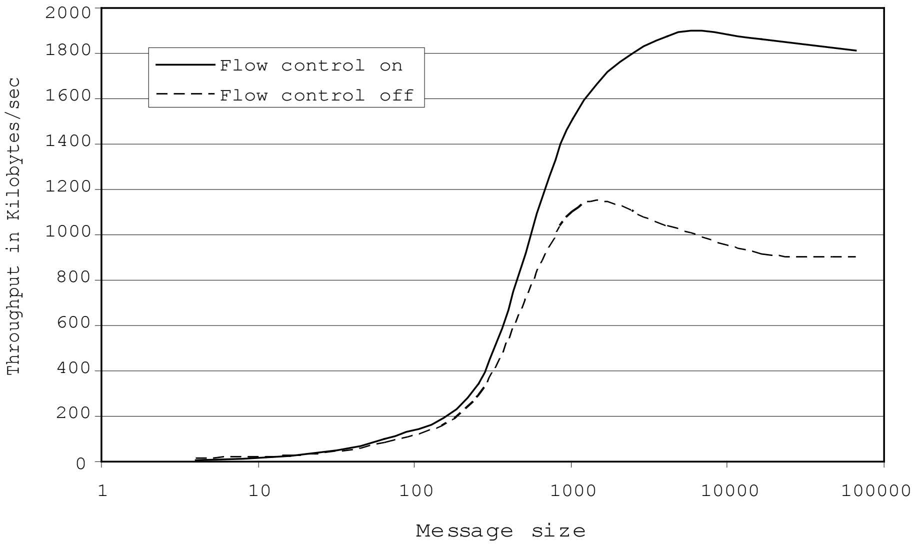


Figure 4.11. The overall fabric throughput in KBytes/sec for the first *front-to-back* with stream based flow-control enabled (solid line) and disabled (dashed line).

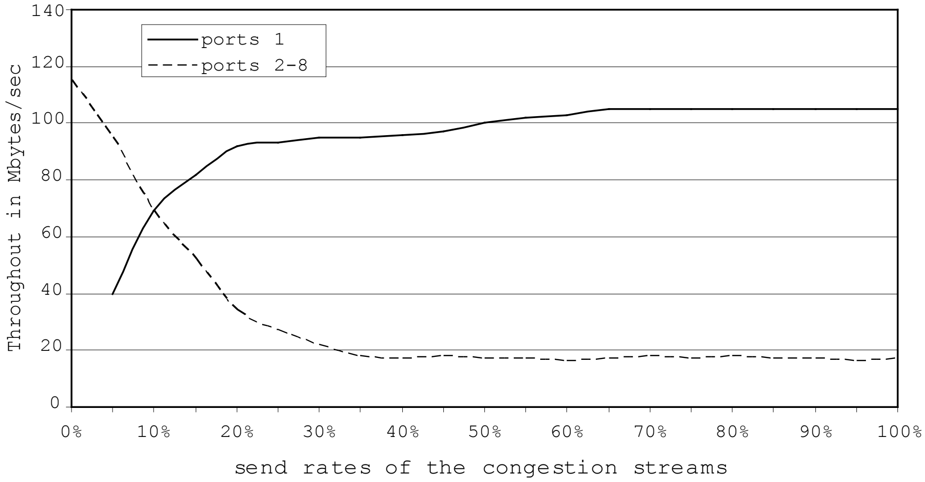


Figure 4.12.Throughput seen at each of the output ports of the switch in the *switch port contention* test with stream based flow-control disabled.

The fourth part of this test investigates the impact of this newly congested stream on the traffic that flow through switch 36 to node 10. The congested streams and the streams targeted towards node 10 originate at the same source nodes. The traffic pattern for the streams to node 10 does not change when they have to share the same links with the congested stream, each runs at 19 Mbytes/sec, delivering 114 Mbytes/sec at node 10.

4.7 Experiments without Flow Control

To examine the effectiveness of the per stream flow control mechanism in GigaNet the tests have been repeated with the source-quench flow control switched off. This does not remove flow control completely as GigaNet also employs a link-level flow control. The results in almost all the tests are identical; as soon as the host interface or a switch port becomes congested this congestion spreads to the other switches in fabric, reducing the overall utilized bandwidth by 50% or more, compared to the bandwidth seen in the case where flow-control was enabled. There was no imbalance noted in the reduction of the throughput over the different streams, suggesting that the scheduling remained fair even under severe congestion. This section presents the results for two tests, *front-to-back* and *switch-port* contention, which are representative for the observations of all tests.

Front-to-back. In figure 4.10 the results of the tests with the first front-to-back configuration are presented for the individual tests, with and without per stream flow control. One observation from the first test was that stream originating at node 22, did not share any links with other streams and as such was able to traverse the whole interconnect without any loss in bandwidth (112 Mbytes/sec). Because this stream does not share any links with other streams it is never subject to congestion and link-level flow control is almost as effective as the per stream flow control. All other flows however are experiencing a reduction in throughput as flow-control is exercised on a per link instead of a per flow basis. The non-discriminatory aspect of link-level flow control effect streams at places where they may not be the cause of congestion. The reduction in throughput for this particular test with a 2048 bytes message size is on the average 27%. In figure 4.11 the overall throughput of the interconnection fabric is presented in relation to message size. As soon as congestion occurs at switches, which is already noticeable at 512 bytes, the overall throughput is reduced. With all streams running at maximum message size the overall fabric throughput is reduced to 58%.

Switch port contention. In this test 7 streams enter and exit the switch through port 2-8. Without any competing traffic each of the port outputs the maximum bandwidth. When to each of the input ports an additional stream is added, targeted for port 1, the effect of link level flow control is visible as soon as these streams start. At 15% send rate the flow control kicks in because of potential congestion at switch port 1.

Because the flow-control operates at link-level instead of individual stream level, it causes all streams coming in over port 2-8 to be equally reduced. The congestion at port 1 now determines the overall throughput of the switch: each stream destined for port 1 runs at 1/7th of the throughput of port 1 (105 Mb/sec), but the other streams are now reduced to run at equal throughput, given the interleaving of cells of both streams at each link, combined with the link-level flow control. This reduces the throughput per outgoing link to 15% of the result with stream based flow control enabled.

4.8 Summary

This chapter detailed the multi-stage interconnect of the AC3 Velocity cluster. A set of experiments was performed to investigate the effectiveness of the flow control techniques employed by the GigaNet switches and host adapters. The results show that the traffic shaping in face of congestion performs very well: hotspot regions do not expand beyond the original switch, no higher order head-of-line blocking could be detected and the resulting balancing between streams competing for bandwidth is fair.

These are very important properties in a production switch were there is no advance control over the communication pattern.

The experiments at the AC3 Velocity Cluster continue, with a focus on the impact of non-uniform traffic patterns, impact of the flow control on message latency, the impact of thousands of competing streams and the impact of burstiness in the traffic sources.

PART – II

Scalable Management Tools for Enterprise Cluster Computing

Introduction

Performance and availability at cost linear to the system size. This premise was the driving force behind the acceptance of cluster computing as a building block for enterprise computing. There was no real technical evidence to support the claim, but the only alternative to *scale-out* to improve performance was to *scale-up* using SMP technology. Although SMPs were very effective in improving performance, using SMPs to improve availability, while technically feasible, could only be achieved at high cost.

The increased reliance of enterprises on powerful computing operations supporting mission critical tasks created a growing commercial demand for improving the performance and availability of the computing infrastructure. The sharp increase in processor and network performance, combined with a dramatic drop in hardware prices, made data-centers of medium and large enterprises ready to grow to thousands of computers. But even though the hardware appeared to be ready to support scalable *rack&stack* clusters, there were still significant software hurdles to take to enable truly scalable cluster computing.

From a software perspective the scalability of the cluster management software is crucial in enabling clusters to scale out. The functionality necessary to support the management of enterprise clusters is often more complex than the applications that will be run on the clusters, which puts tremendous pressure on the scalability of the components that need to provide the core of the distributed operations. These components also need to be modularized in such a way that they become building blocks that can be used to create services that are as scalable as the core components.

Historically cluster management systems were developed by cluster hardware vendors, and the scalability of the management system was limited to the particular platforms for which they were developed. *VaxClusters, Sysplex, NonStop* [58,70,53], all used software components that were scalable to tens of nodes at best, which was sufficient to support the cluster products offered. The only scalable solution available was the cluster management system for the *IBM SP2* series, which was capable of managing up to 250 nodes using a voting-based distributed systems technology [3].

Classifying cluster applications

Use of the traditional management technology for controlling the large-scale enterprise cluster computing systems revealed the following problem: Each of the commercial cluster management systems was developed with a particular application domain in mind. The use of clusters in enterprise data-centers brought in a wide variety of new applications and, none of the original management technologies provided an overall solution or a general framework with which a coordinated approach could be taken for managing such a diverse set of services.

One way to classify the new applications is to examine the structuring techniques used to achieve scalability. Two main approaches can be distinguished through which scalability on clusters is achieved: through *cloning* and through *partitioning*.

Cloning is used for applications that can be replicated onto many nodes and where each node has access to the identical software and data set. This allows client requests to be routed to any of the clones, with load-balancing as the main criteria for assigning client-server relations. Web server-farms are an example of cloned servers, and the premise is that under load additional nodes can be added to handle the increased demand. Similarly, many enterprise parallel computing operations can be improved by adding nodes to a pool of “workers”. In general the cloned approach not only provides improved performance but also improved availability of the application.

Partitioning is used for those applications that have a significant state to maintain and where cloning the state would be impractical because of the overhead involved with keeping the replicated state consistent. Instead the application data is partitioned over the nodes in the cluster based on application specific criteria. For a database the partitions may be based on layout of the tables and the relations between tables combined with the expected request patterns. Middle-tier application-server systems divide tasks based on customers grouping, or a collaboration server will group participants together based on real-time assessment. In general the partitioned application does not provide improved availability without the use of additional technology.

A second way to classify cluster applications is to examine their use of shared storage. Both cloned and partitioned servers can be served from local storage or from a shared storage infrastructure, usually in the form of network attached storage devices. Cloning the data to local storage requires a strategy for ensuring the consistency of the replicas with the master copy. Using shared storage, combined with cloning the application can be seen as a hybrid approach that is used in cases where the updates to the state are too frequent to warrant a fully cloned approach, but where a large number of cloned application nodes is needed to support scalability. An example is a large E-mail farm, where any of the application servers can handle client requests accessing the data from central storage. The E-mail server cluster is a particular example as the partitioned approach can also be used, either with or without shared storage. In the partitioned case the mailboxes are distributed over the nodes in the clusters, based on local access to the data.

Keep in mind that without shared storage a node crash in the partitioned case results in the loss of availability of part of the data, unless the system makes use of other application level replication techniques such as hot-standbys.

Evaluation of an enterprise-class cluster management system.

To better understand the problem space of building management systems for these kinds of application clusters, an analysis was performed of one of the more modern cluster management systems. The *Microsoft Cluster Service* (MSCS) was first released in 1997 and specifically targeted the application fail-over market. In a fail-over clusters the applications from a failed node can be restarted at the remaining nodes in the cluster. Soon after its first release I lead a team of academics and Microsoft research and product engineers to perform an in-depth analysis of the system which resulted in a publication at the IEEE Fault-Tolerant Computing Symposium (FTCS) in 1998 [100].

Although MSCS was successful in the market it was targeted at, the main question remained whether it could scale beyond the initial 2-4 nodes on which it was shipping.

There were no experiences with running MSCS at larger installations so I adapted the software to run on a 32 nodes cluster at Cornell. The investigation focussed on the scalability of the core distributed components: the cluster membership service and the global update protocol. The results from these experiments were published at the second Usenix Windows NT Symposium in 1998, and are presented in Chapter 7.

Lessons learned

One of the conclusions of the scalability experiments with MSCS was that the transparent use of distributed systems technology, without taking into account the specific properties of the technology, yielded a system that did not scale. Under optimistic conditions the system could function with 10 nodes or more, but any realistic load on the nodes would completely degradate the ability of the system to perform updates or make membership changes. This degradation was not caused by the particular distributed algorithms used to maintain membership and perform updates but by the choice of standard RPC to implement the algorithms. Originally the algorithms were developed to be executed over a high-performance broadcast bus and in the MSCS case they were executed using repeated RPC calls, which very quickly became a scalability bottleneck in the system.

These conclusions resonated with our earlier experiences of building advanced distributed systems software. Our systems were used in a variety of complex production environments, such as the stock-exchanges, air-traffic control and large military systems. A decade of building software systems that transitioned into the real-world thought us a number of valuable lessons. Many of our assumptions about the real-world and how software was being used had been proven to be too idealistic. The software was used in every possible situation, except for the ones that it was originally intended for. These lessons have been very valuable and have become a driving factor in our thinking about how to structure scalable distributed systems. These lessons have been collected in a paper titled "*Six Misconceptions about Reliable Distributed Computing*" which first was published at the 1998 ACM SIGOPS European Workshop and later at the 1999 International Symposium on High Performance Distributed Computing. The lessons are summarized in chapter 5.

Applying the lessons

These conclusions drove much of my research agenda for 1997 and 1998 and I architectured a new system, dubbed *Quintet*, in which distribution in all its aspects was made explicit. Quintet provided high-level tools to help the application server programmer with operations such as replication, but the developer decided, what, where and when to replicate. Some of the management and support tools, such as a *shared data structure toolkit*, relied on generic replication, but the developers that use them are aware of the implications of importing this functionality. An overview of Quintet can be found in chapter 6.

A framework for scalable cluster management

Using the experiences with traditional cluster management systems and the lessons learned from developing scalable distributed components, I developed the *Galaxy Cluster Management Framework*. Galaxy is a scalable solution for the management of clusters in large data-centers. Its multi-tier management infrastructure provides the ability to manage compute farms at multiple locations, with within a farm islands of specialized clusters that are managed according to a cluster profile. A profile describes the distributed systems services needed to support the operation of the particular application cluster. The resulting architecture was first published in a paper at the 2000 IEEE International Conference on Cluster Computing. Galaxy is described in detail in chapter 8.

Continuing the quest for scalable cluster management

The world of enterprise cluster computing remains very much in flux. The experiences with the demise of many e-commerce operations combined with sky-high energy prizes have switched the focus of cluster management from optimal performance to optimal energy usage. New computing devices based on *blade* architectures promise lower power consumption if managed correctly.

Using the small blade architectures it is likely that the number of computing devices in data centers will continue to increase, which will increase the pressure on the scalability of the management system. One area that has only receive limited attention over the past years is that of fully automated data-center management. The drive will be to achieve *autonomic* systems that can be self-healing and self-reconfigurable, probably based on complex rule-based systems.

At the other end of the spectrum we find the new architectures for high-performance servers that are build out of distributed components based on Infiniband or ServerWorks hardware. These components are combined with intelligent storage architectures that have significant processing power close to the disks itself. These deeply distributed architectures introduce many new challenges for the scalability of the cluster management systems for the years to come.

Chapter 5

Six Misconceptions about Reliable Distributed Computing

This chapter describes how experiences with building industrial strength distributed applications have dramatically changed the assumptions about what tools are needed to build these systems.

5.1 Introduction

For the past decade the reliable distributed systems group at Cornell has been building tools to support fault-tolerant and secure distributed systems [7]. These tools (Isis, Horus and Ensemble) have been successful in an academic sense because they functioned as research vehicles that allowed the community to explore the wide terrain of reliability in distributed systems. They were also successful commercially as they allowed industrial developers to build reliable distributed systems in a variety of industrial settings. Through our interactions with industry we learned a tremendous amount about the ways distributed systems were built and about the settings in which the Cornell tools were employed.

What we found is that professional developers have applied these tools in almost every possible way… except for the ways that we intended. By building major distributed applications such as the New York & Swiss Stock Exchanges, the French Air Traffic Control and CERN's data collection facility, the professional development community revealed that tools of the sort we offered are needed and yet that our specific toolset was not well matched with the requirements of serious practical builders [9]. This was not because the technology was somehow "incorrect" or "buggy" but simply because of the particular ways in which distributed systems of a significant size are built. These surprising experiences led us to reevaluate our tools and how we can improve support for building reliable distributed systems.

In this chapter we want to discuss the misconceptions we had about how distributed systems are built. We believe that many others from the research community shared our assumptions, and that they still are the basis for new research projects. It is thus important that others learn from our mistakes

5.2 The misconceptions

The design of interfaces, tools and general application support in our systems was based on beliefs concerning the manner in which developers would build distributed applications under ideal circumstances. This methodology presumed that only the lack of proper technology prevented the developer from using, for example, transparent object replication. It is only now that the tools capable of supporting this kind of advanced technology have been built and used by the larger public, that we have come to recognize that some of the technology is just too powerful to be useful. We offered a Formula One racing car to the average driver.

In this chapter we describe some of the more important idealistic assumptions made in the earlier stages of our effort. In retrospect, we see that our assumptions were sometimes driven by academic enthusiasm for great technology, sometimes overly simplistic or sometimes just plain wrong.

5.2.1 Transparency is the ultimate goal

A fundamental assumption of our early work was that the developer seeks fault-tolerance and hopes to achieve this by replacing a critical but failure-prone component of an existing system by a high reliability version, obtained using some form of active replication [8]. Thus although we considered client/server transparency as a potential source for problems, we also believed that replication transparency from the point of the server writer was a laudable goal. Developers should not have to worry about replication strategies, failure conditions, state transfer mechanisms, etc., and be able to concentrate on the job at hand: implementing server functionality.

It turns out that server developers *want* to worry about replica configuration, intervene in failure detection or enabling explicit synchronization between replicas. There was only a small class of server applications where the designer did not care about the impact of replication, and most of these involved server replicas that needed no access to shared resources and were not part of a larger execution chain.

The majority of systems in which transparent replication is used became more complex, suffered reduced performance and exhibited potential incorrect behavior traceable to the lack of control over how replication is performed. There is a strong analogy with starting additional threads in a previously single threaded program, where the designer is not aware of the added concurrency.

5.2.2 Automatic object replication is desirable

The prevailing thinking over the past decade has been that object systems represent the pinnacle for transparent introduction and presentation of new system properties (components did not exist until recently!). Objects provided an unambiguous encapsulation of state and a limited set of operations. By using language or ORB features we could automate the replication of objects without the need for any changes to the objects, while using state machine replication. Products such as Orbix+Isis [60] and projects such as Electra [63] have been successful in a technical sense in that they succeeded in implementing these techniques, but their practical success in the hands of users was very limited.

In addition to suffering the transparency limitations just described, automatic object replication exposed problems in the area of efficiency. Apart from the fact that the state machine replication was a very heavyweight mechanism when used with more than two replicas, a generic replication mechanism needs to be conservative in its strategies and may be very limited in terms of available optimizations. When allowing the developer control over what and especially when to replicate, optimizations can be made using the semantics of the application.

5.2.3 All replicas are equal and deterministic

One of the more surprising violations of our early assumptions was that many services did not behave in a deterministic manner. The systems we were confronted with were often complex multithreaded or multi-process systems, running on several processors concurrently, containing massive subsystems designed and built by different teams. Trying to isolate a generic template for transforming such systems into reliable ones using a toolkit approach turned out to be impossible, because generic solutions invariably depend upon strong assumptions about deterministic behavior.

5.2.4 Technology comes first

It makes good sense for an academic research group to focus on a technology for its own sake. After all, academic research is rewarded for innovation and thoroughness. As tool developers, however, we need to focus on the application requirements first and then use technology only if it matches. When an academic group transitions technology into commercial use it suffers from a deep legacy of this early set of objectives. Obviously, there are success stories for academic-to-commercial transition, but they rarely involve general-purpose tools. In particular, academics tend to focus on the hardest 10% of a problem, but tool developers for industrial settings need to focus first on doing an outstanding job on the easiest 90% of the problem, and dealing with the residual 10% of cases only after the majority of cases are convincingly resolved.

5.2.5 The communication path is most important

Many of the reliability techniques add extra complexity to the system design and impose performance limitations. We spent years building super-efficient protocols and protocol execution environments, trying to eliminate every bit of overhead, and sometimes achieving truly dazzling performance. Developers, however, are less concerned about state-of-the-art performance: their applications are heavyweight and the processing triggered by the arrival of messages is significant. The parts of the system they stress more and where performance really mattered is that of management. Consistent membership reporting, bounded-delay failure handling, guarantees on reconfiguration and bounds on the costs of using overlapping groups are the issues they truly care about. Moreover, to slash performance we often accepted large footprints, in the form of code bloat from inline expansion of critical functions and increased memory for communication buffering. The user often favors a smaller solution at the cost of lower performance.

5.2.6 Client management and load balancing can be made generic

We taught the user to replicate critical servers, then load balance for high performance, and we argued that it could be done in a generic, very transparent manner. Indeed, this fits the prevailing belief, since many RPC systems work this way (consider cluster-style HTTP servers). But the situation for replicated servers proved much more complex as soon as clients had a longer lasting relation with a server instance, with state maintained at the server. In those cases client management could no longer be made generic, without resulting to protocol specific tricks or using mechanisms that reduce performance significantly. Another aspect was that server developers wanted to have strong control over the distribution of clients among the servers. They almost always were interested in using application-specific knowledge to group client connections together to maximize server-processing efficiency.

5.3 Conclusion

What conclusions can we draw from our experiences? The central theme in most of the erroneous assumptions was that of *transparency*. Trying to achieve transparent insertion of technology, to augment services with reliable operation, seemed a laudable goal. Looking back we have to conclude that transparency has caused more pain and tragedy than it has provided benefits to the programming community. We believe the conclusion is warranted that we need to avoid including strong transparency goals in our future work. Similar conclusions can be drawn from experiences by other groups that are currently evaluating their research results [46].

This conclusion has driven a new research agenda and we have developed a new system, dubbed *Quintet*, in which distribution in all its aspects is made explicit. Quintet has high-level tools available to help the programmer with operations such as replication, but the developer decides, what, where and when to replicate. Some of the management and support tools, such as a *shared data structure toolkit*, rely on generic replication, but the developers that use them are aware of the implications of importing this functionality. A detailled description of Quintet can be found in the next chapter.

The decision to give full control to the developer is in strong contrast with the current trends in reliable distributed systems research, where transparency is still considered the Holy Grail. These new systems [68,93,104] experience the same limitations that the serious use of our systems exposed. Only by restricting the useable model will these systems be able to support developers in a consistent manner. Quintet does not restrict the traditional programming model in any way and while providing the developer with more effective tools to do his/her job.

Chapter 6

Quintet, Tools for Reliable Enterprise Computing

This chapter describes Quintet, a system for developing and managing reliable enterprise applications. Quintet provides tools for the distribution and replication of server components to achieve guaranteed availability and performance. It is targeted to serve the application tier in multi-tier enterprise systems, with components constructed using Microsoft COM. Quintet takes a radical different approach from previous systems that support object replication, in that replication and distribution are no longer transparent and are brought under full control of the developer.

6.1 Introduction

In enterprise settings computing systems are becoming more and more organized as distributed systems. These systems are critical to the corporate operation and a strong need arises for making these systems highly reliable. The first step in addressing these needs has been taken by industry: based on their experiences with dedicated cluster environments, as new cluster management software systems have been developed that target off-the-shelf enterprise server systems. In general commercial cluster products provide functionality for the migration of applications from failed nodes to surviving nodes in the system. Although this offers some relief for systems such as web servers, databases or electronic mail servers, it does not facilitate the development of systems that are capable of exploiting the cluster environment to its full potential.

Quintet addresses some of the problems that arise when building reliable distributed enterprise applications. The system provides development and runtime support for components that make up the application tier of multi-tier business systems. In the target systems this tier is constructed out of servers build as collections of COM components. Components developed using the tools provided by Quintet are able to guarantee reliable operation in a number of ways, and the system is extensible in that new guarantees and interfaces can be added.

The project is concerned with research into two areas:

- What development tools are needed to build reliable distributed components for enterprise computing? With a strong focus on efficiency, simplicity and ease of use.
- What infrastructure is needed for reliability management on the high performance cluster systems providing the component runtime environment?

The next two sections in this chapter provide some background on the way Quintet addresses issues surrounding reliability and distribution transparency. This is necessary to understand the design choices that have been made. The section following the introduction provides an overview of Quintet's functionality and the solutions that can be built with the Quintet tools. After a description of the target environment and relation between Quintet and the Microsoft Transaction Server, the chapter provides details on the major system components that make up Quintet.

6.2 Reliability

Component reliability in Quintet addresses two aspects of distributed computing: high-availability and scalable performance. The first is concerned with that given a limit to the number of node failures, the system guarantees that the remaining set of nodes continues to provide the required functionality. The second aspect ensures that the system, using adaptive methods, distributes the load over available resources to guarantee optimal performance.

Reliability in Quintet is described using a Quality-of-Service specification. When a new component is added to the system, the administrator describes the reliability requirements of the component, which are input for the runtime system and for the component class factories. The specification can be changed on-line and the system can be requested to reconfigure accordingly.

A simple approach for providing high-availability and scalable performance would be to replicate components over several server nodes and to provide client fail-over and load balancing to achieve the reliability goals. Although this is an approach that certainly can be used in Quintet, several more tools to design the distribution of server components, beside active replication [8], are offered. The designer has a full range of synchronization, replication, persistency, data sharing & consistency, checkpoint & logging, coordination and communication tools available to construct components that are distributed in a fashion that exactly match its reliability requirements.

6.3 Transparency

A decade of building large distributed systems in industrial settings (see chapter 5) has shown a serious mismatch between the available tools for constructing reliable systems and the requirements of professional system builders. Many of the problems can be reduced to the fact that tool-builders were trying to achieve the transparent insertion of their technology, while system-builders need full control to achieve acceptable performance or efficient management.

The pinnacle of transparent operation can be found in the attempts to provide fully automatic object replication. By using language or ORB features the replication of objects could be automated without the need for any changes to the objects, while using state machine replication. Products such as Orbix+Isis[60] and projects such as Electra [63] have been successful in implementing these techniques, but their success in the hands of users was very limited.

In all observed systems (see chapter 5) it turned out that server developers *want* to worry about replica configuration, intervene in failure detection or enable explicit synchronization between replicas. There was only a small class of server applications where the designer did not care about the impact of replication, and most of these involved server replicas that needed no access to shared resources and were not part of a larger execution chain. The majority of systems in which transparent replication is used, become more complex, and suffer reduced performance and potential incorrect behavior. This is traceable to the lack of control, within the application, concerning how replication is performed. There is a strong analogy with starting additional threads in a previously single threaded program, while the designer is not aware of any concurrency.

In addition to the transparency limitations just described, automatic object replication exposed problems in the area of efficiency. Apart from the fact that the state-machine replication is a very heavyweight mechanism when used with more than two replicas, a generic replication mechanism needs to be conservative in its strategies and may be very limited in terms of available optimizations. When allowing the developer control over what and especially when to replicate, optimizations can be made using the semantics of the application.

These observations have resulted in that in Quintet distribution in all its aspects is made explicit. Although there are many tools available to help with operations such as replication, the developer decides, what, where and when to replicate. Some of the management and support tools, such as the shared data structure toolkit, rely on generic replication, but the developers that use them are aware of the implications of importing this functionality.

The decision to give full control to the developer is in strong contrast with the trends in reliable distributed object research, where transparency is still considered the Holy Grail (chapter 5). These systems [68,93,104] experience the same limitations that the serious use of our systems exposed. Only by restricting the useable model will these systems be able to support developers in a consistent manner. Quintet does not restrict the traditional programming model in any way and provides the developer with more tools to do his/her job.

6.4 Quintet goals

Quintet targets the development of client/server computing in multi-tier enterprise systems, where there are reliability requirements for the servers. In the prototype system, the servers are implemented using Microsoft COM component technology.

The central research goal of Quintet is to find the collection of tools that is most useful for the developers of reliable components. Given that this is not an area where past experience can drive the selection of these tools, the project was started with building a limited set of essential tools and interfaces. Iterative, based on user feedback, the tool collection was changed to meet the real needs. One of the major reasons for targeting COM based server environments was that these have a perceived need for reliability and the project is very likely to get valuable feedback from the user community to ensure the much needed improvement cycles. An overview of the initial tool set appears in a later section of this chapter.

The server components developed with Quintet are COM aware as all services offered are only available through COM interfaces. The expectation was that the majority of client/server interaction in this environment is DCOM based, but there is nothing in the system that inhibits client/server communication based on RPC, HTTP-tunneling, sockets or to integrate an IIOP-bridge. What ever client/server access mechanism is used, the actual application tier components are implemented as COM classes and instantiated through COM class factories.

In a traditional DCOM client/server system the event that triggers the instantiation of a component in that of a *CreateInstance* call at the client system. For reliable COM server components the rules for instantiation can be more complex and are often based on the reliability QoS specification for the particular component. For example server components can out-live client connections to ensure real-time volatile state replication, where the component is only made persistent and decommissioned after no new client connection was made within a given time period.

In the current set of tools offered in Quintet it is assumed that instances of the same component have a certain need for cooperation. The basic communication tool is pre-configured to provide a component with primitives to communicate with all other instances of the same component and to receive membership style notifications. Quintet based replicated components are not forced to maintain identical state, the developer chooses when and what to replicate, to which components. How optimistic (or pessimistic) the state replication strategy is, depends on application trade-offs, and can be adjusted on the fly. Cooperation in Quintet is not limited to components of the same class. Different components, can transfer state, synchronize, vote for leadership, use shared data structures and use the communication tools in explicit manners.

Given that all distribution is explicit, a major concern in Quintet is that the exposed complexity could make the development task more hazardous, yielding systems that are more error prone and thus implicitly defeating the reliability goals. The tools and interfaces are designed with care to match the existing COM programming practices as much as possible, making the transition for developers as simple as possible. In good Windows tradition, a number of Programming Wizards are provided to assist in the more complex tasks.

The second goal of Quintet was to build an efficient runtime environment to support the development of complex tools. In Quintet new algorithms for scalable lightweight object membership, fast distributed synchronization, efficient component migration, have been applied. The Core Technology (QCT), which implements the underlying communication system, is designed with high-performance cluster communication interfaces in mind.

6.5 Relation with MTS

Although the Microsoft Transaction Server (MTS) is concerned with offering solutions to server components with a different set of requirements, Quintet has in its implementation some solutions that are similar to MTS. The way the component management service is the container server for the components it manages and the way it maintains contexts for each component instance are similar to the way MTS manages its components. The similarity is based on that this is the correct way of managing COM objects.

Two other mechanisms in Quintet have identical counterparts in MTS: Security is implemented using a role based management system and long running components can be temporarily *retired* without notifying the connected clients. The role-based security was chosen based on a research decision and its similarity to the MTS solution can be seen as accidental. The *retire* operation was added to Quintet, based on the argumentation by the MTS architects that memory consumption by long running components is the limiting factor in scaling component servers such as MTS and Quintet. We do not have any experiences that support this claim, but the arguments seem reasonable and by implementing the facility Quintet can be used to research this issue.

6.6 Target environment

Quintet is designed to function on a collection of server nodes, organized into a cluster, with some form of cluster management software offering basic services such as node addressing, node enumeration, object naming and basic security. The prototype implementation of Quintet uses the Microsoft Cluster Service (MSCS) [100], LDAP accessible naming service (Active Directory) and the standard Windows NT/DCOM security mechanisms (LanManager).

The Quintet implementation assumes cluster sizes of 4 to 16 nodes. Although nothing in its design prohibits the use of larger sized clusters, the distributed algorithms used in the Core Technology are optimized towards clusters of this size. The implementation is modular in the sense that the Core Technology components can be replaced if the need for that arises. A fundamental assumption in the construction of the system is that the intra-cluster communication can be performed an order of magnitude faster than the client/server interaction.

Although a first concern of Quintet is correctness of the services it offers, providing scalable performance is an important second goal. In a related measurement project MSCS and DCOM are thoroughly analyzed to understand the performance boundaries of these technologies (see chapter 7), and to be able to offset Quintet introduced overhead and costs correctly.

6.7 System overview

Components developed with Quintet are available on the server nodes through application servers (Quintet Component Manager) that are configured to export the components through the traditional component registration channels. Instantiation requests arrive at the servers, which are responsible for the loading and unloading of class factories, and tracking component instances.

A variety of different styles can be used in developing reliable components, all depending on the reliability requirements of the application. Components can be longer running, actively replicated components, where each new client connection only triggers the instantiation of some client state. Or each new instantiation request can result in the creation of two instances at different nodes that collaborate in a primary/backup fashion.

In principle the class factories implement client management and replica instantiation, while the components implement the replication strategy. In implementing each of these tasks the developer is assisted by Quintet functionality. Quintet provides default implementations for general cases.

The system implementation consists of sixmajor building blocks

1. *Core Technology.* The communication system on which the component manager and the component runtime are based. It provides membership and multicast communication functionality.
2. *Server Component Management*. Provides the registering and loading of the server components. Manages component placement, fault monitoring and handling, security and basic system management.
3. *Server Component Development*. The basic tools for the developer to construct the server components
4. *Server Component Runtime*. Tool implementation and management, is part of the component manager.

5. *System Management Tools*. A collection of tools for administrators to monitor and manage the system and its individual components.
6. *Client Runtime*. Mechanisms to support connections to potentially replicated components by regular DCOM clients. Support for failover to alternative component instances upon failure.

Each of the different building blocks is described in detail in following sections.

6.7.1 Core Technology

Quintet Core Technology (QCT) is the basic building block for the server management and component runtime. It is a lightweight implementation of a *Group Communication Service* [7], specifically targeted towards high-performance clusters. It uses MSCS style addressing and makes use of some of the nodes management features of the MSCS management software [100]. It uses this information to locate other Quintet component managers, and to determine which network interfaces to exploit for intra-cluster network communication.

QCT is designed to run over both reliable and unreliable interconnects and is optimized towards user-level communication interfaces such as VIA [33] and U-Net [37]. The low-level message handling interfaces make extensive use of asynchronous message transfer and facilities such as Windows NT completion ports to optimize interaction with the network.

QCT offers *Virtual Synchrony* guarantees [7] on its communication primitives, ensuring the ordering of messages in relation to membership changes and *atomicity* on all message delivery. The communication interface provides a multicast primitive to send all members in a group and a *send* primitive to address a single member. Messages sent with the multicast primitive can be sent with either the basic guarantees (atomicity) or can be extended with a *total order* guarantee ensuring that all members see all messages in this group in the same order.

QCT provides an internal interface, mainly used by the component managers (see next section). The components and class factories see a higher level interface for communication. To make the system scalable and not overuse the heavy weight virtual synchronous membership for each instantiation of a component, a lightweight component membership mechanism is layered over the basic system.

Each component is automatically a member of its *ClassGroup*, which provides membership notification and communication to all instances of a single component class. All the class factories of the same component class, present at the different component managers see membership change notifications whenever a component is instantiated or destructed. The class factories also see membership changes whenever a new instance of the particular class factory joins the system. The components only see changes in the component membership, not of the factories, and the components only receive membership updates if they explicitly register for it. The virtual synchronous membership agreement algorithm is only run in case of the failure of an object manager, or when a class factory at a component manager is unloaded.

Components can make use of the group communication facilities outside of the ClassGroups interface by using self-defined groups and names. The component can choose to either use the lightweight component membership or the more heavyweight low-level QCT interface.

6.7.2 Component Management

The *Quintet Component Manager* (QCM) is the central unit in the management of the reliable components. The functionality of the manager includes: loading of component libraries, starting of class factories, performing security checks, client administration & configuration, failure handling, dynamic load management and system administration. The manager contains the Core Technology and the runtime for the tool collection.

QCM is registered at each server node to implement all the component classes it manages, resulting in that the Service Control Manager at the node routes regular DCOM instantiation requests for the components to QCM. The method by which a client receives information about which node to contact for its instantiation request depends on the particular client technique used, which are described in the section on the client runtime. The class factories for the requested component are expected to collaborate on providing a hint to the QCM at which node the instantiation is preferred. This information is relayed to the client moniker object or the proxy process. If the class factories suggest “don’t care” the QCM makes a decision based on the QoS spec for the component.

For each component instance the QCM maintains a *shadow* object (context object in MTS terms), where the object references, returned to the client, refer to. The shadow object contains administration, statistical and debug information. Longer running components, with a low method invocation frequency, can be requested to persist their state and then destruct themselves. At the next method request, directed to the shadow object, the component is reloaded from the saved state. This mechanism can not be used for all types of components, as for example components engaged in active replication can not be decommissioned.

It is possible to migrate active components to other nodes in the system, and there are two mechanisms from which the component state at the new node can be recreated:

1. The component can implement an IMigrateState interface, or
2. The manager can forcibly use the checkpoint and reload mechanism. Requests from clients that are not yet updated with the new location of the migrated component are forwarded based on indirection information in the shadow object. The shadow object is garbage collected after the original component is destructed.

Each node in the cluster runs one or more component managers. How many managers run at a node depends on the particular component configuration. If a component is considered to be unreliable, for example during a development phase, its potential failure is isolated from the rest of the component system by running the component in a separate address space attached to a private manager. If a component crashes it will not cause the failure of the other components. For scheduling and network efficiency it is desirable to have a single component manager per node, controlling all components at that node. The first QCM process runs as a Windows NT service, any additional processes are started by the first QCM process. The primary QCM process is under control of an MSCS resource monitor to ensure automatic restart in case of failure.

6.7.3. Component Development Tools

Quintet offers a collection of tools for the developer to use for the construction of the class factories and the components. The following is a short list of the most important tools.

- *Basic membership & communication*. As already described in the section on Core Technology, each component and its class factory are automatically a member of its *ClassGroup*. The components can register for receiving membership change notifications, and additional interfaces are available to query the membership through the runtime and from components such as the *shared data structures*. Next to the use of the ClassGroup the component is free to create, join and leave other groups and communicate using those groups. The component, however, cannot leave its ClassGroup other than through destruction.

- *State maintenance*. A component can implement a shared state interface and register this interface with the object manager. State update can be performed manually by a component notifying the component manager that it now wants its state transferred to all other components that have registered the same component state interface. If needed, any synchronization before the state update is to be handled by the component itself. The component manager retrieves the state from requesting component and updates all components in total order.

  The developer can also choose for an automated version of state update. Components notify the QCM whenever the state has changed, and whenever they are in a position to receive the state update. As soon as all components have signaled their availability the component manager updates the state. Any conflict resolution needs to be implemented by the component state receive routines. An example where this kind of automation is useful is when using a primary component with a collection of hot standbys.
- *Shared data structures*. To support simplified state sharing strategies Quintet offers a collection of data structures (hash table, associative sets, queues, etc.) that can be shared among component instances. The object managers implement the runtime for this and ensure that component instances which share interface references to a shared datastructure, always have access to a local copy. Updates to the data structures are guaranteed to keep all replicas in a consistent state.
- *Voting*. A component can propose an action, on which the participating components vote to accept or reject. More complex algorithms for quorum techniques, barrier synchronization, distributed transactions and leader election are implemented using this basic voting interface. This is similar to the services in [3].
- *State persistence*. Quintet offers a persistent object store to the components from which the components can be initialized. The mechanism is used for crash recovery, system startup and the decommissioning and restart of long running components. Components are not automatically persistent, they use a checkpoint and logging interface to explicitly persist their state.

6.7.4 Component Runtime

Many of the tools Quintet offers to the developer have a significant runtime component to them. The majority of the tool collection is implemented using the facilities offered by QCT. With the exception of some management modules, the communication between tool instances runs over the same heavyweight communication endpoint as the *ClassGroup* of the component that uses the tool. The lightweight addressing space is divided such that class factories, component instances and tools can be addresses separately and messages can be multicast to the appropriate subsets. This sharing is an optimization, when the runtime detects that a tool instance is use by different components, it creates its own heavyweight endpoint.

An example of a tool with a major runtime component is the one that implements *shared data structures*. The runtime implements the data structures itself, the distributed access, the consistent updates, and the replica & location management to ensure that components always have local read access.

To ease the development of new tools, a small support toolkit was built that implements basic data types, message handling and the serialization of basic data types through QCT.

A tool that does not make use of the QCS facilities is the *Persistent Object Store*, which uses the checkpoint and logging facility offered by MSCS [100]. This ensures that the data is always available to the nodes that are active in the cluster as MSCS terminates minority partitions that have no access to the shared *Quorum resource*. The Quorum resource is a shared disk, which also stores the checkpoints and logs.

6.7.5 System Management

Experiences show that complex server systems such as Quintet are only as useful as the system management tools and interfaces that accompany it. Without these tools the system becomes painful to use, difficult to monitor and diverts the attention of the developer from the most important task: developing robust components.

The key system management tool is a traditional Win32 explorer style application, which can be used for all the administrative tasks. The management tool is developed as a traditional COM client/server application, with the server functionality implemented in the Component Managers. There are command line counterparts of the tool, but they are geared towards the use in shell scripts. Several tool instances can be running at the same time. The tools are augmented with failure detection mechanisms outside of the scope of COM to ensure timely failover to another component manager, in the case of a manager or node crash. The RPC timeout mechanism in COM is very tolerant but a 30 seconds delay is unacceptable for Quintet purposes.

The management tool provides developers with the ability to add components to cluster through drag and drop, and to add a Reliability QoS specification as well as the security information for the new component. The tool can be used to control manager and component configuration, and runtime control tasks, such as component migration, can be performed manually.

The Component Managers contain several methods to monitor the operation of the system and to monitor components on an individual basis. Information ranging from statistics to individual method invocation and results can be monitored and displayed in the management tool.

The tool can be extended on a per-component basis with component specific control and management functionality (see the section on extensibility).

6.7.6 Client Runtime

Client access to the component instances is still a major research issue. Currently two approaches are implemented: In the first approach the client is failover aware and uses a reconnect mechanism to hookup with an alternative component instance. The second approach leaves the client unaware of the new situation and uses a local proxy process through which all calls to the component replicas are routed.

Although prototypes of both approaches are implemented and running, the difficulty is in determining whether all cases are handled correctly. Realistic COM client/server applications often have very rich interaction patterns and some of the pre-built MS components implemented using the ATL or MFC toolkits introduce additional levels of complexity.

The failover aware approach has its limitations as COM insists on making distribution of the component fully transparent and treats each component as if it is local, providing no failure information about the distributed case. Quintet's support here consists of a set of moniker objects that internally interact with the component managers to connect to a selected node and are also responsible for the selection of new nodes after failure occurs. The developer in the design of the client needs to catch these failures and use the moniker to reconnect. This approach has only been made to work in C/C++ COM environments and have not found their counterpart (yet) in VB and Java.

The most promising of the two approaches is the one that uses a local proxy to implement the client/server interaction. The proxy receives component information from the component manager and registers itself at the local node as implementing these components. The Service Control Manager at the local node then routes all CreateInstance requests to the local proxy. The proxy interacts with the Component Manager to create the correct forwarding path and upon server failure or component migration adjusts the path accordingly. The proxy is efficient in that it inspects the stack of the incoming RPC call to do some cut-through routing without the need to implement the interface locally and fully execute each RPC invocation. As always Connection Points (COM's version of callbacks) make the whole mechanism more complex, and are handled with support from the Component Managers. The Managers keep track of this kind of full duplex connection, and ensure that upon client handoff to another component instance, the component that now handles the client has the right client routing information.

6.8 Extensibility

The base system is extensible by developers in a number of ways:

1. Component specific management modules can be added to the management tools.
2. Component specific debug support can be added to component managers.
3. Runtime support for new tools can be added to the component managers.
4. General component management can be added to the component managers.

The extensions to the management tools and the component managers are implemented as COM components and loaded on demand though the regular configuration information in the NT registry. Each extension type needs to implement a predefined interface through which the component can be initialized and given access to environment it executes in.

To enable the development of these extensions, the component manager and the administration tool export a set of interfaces that can be used by each extension to implement its functionality.

6.9 Future Work

There are a number of major research issues that have arisen but have not been worked out yet. The most important issue is that of inter-component dependencies, the obvious solution seems to hint to the use of component groups and manage the groups as single units, but as yet it is uncertain what the best way is to express the dependencies and how to manage them at runtime. Which approach to client runtime is the right one, remains an open issue until there is more experience with actual deployment of the system and more industrial-strength application development has been done.

Chapter 7

Scalability of the Microsoft Cluster Service

7.1 Introduction

An important argument for the introduction of software managed clusters is that of scale: By constructing the cluster out of commodity compute elements, one can, by simply adding new elements, improve the reliability of the overall system in terms of performance and in availability. The limits to how far such a cluster can be scaled seems to be dependent on the scalability of its management software, which in its core has a collection of distributed algorithms to guarantee the correct operation of the cluster. The complexity of these algorithms makes them a vulnerable component of the system in terms of their impact on the overall scalability of the system.

This chapter examines two of the distributed components of the Microsoft Cluster Service (MSCS) [100] that are most likely to have an impact on its scalability: the *membership* and the *global update* managers. The first sections of the chapter will provide some general background on these distributed services and related scalability issues. After this introduction the algorithms used to implement these service are described in detail and an analysis of their impact on scalability is presented. The scalability analysis is based on an off-line analysis of the algorithms as well as the results of on-line experiments on a cluster with a, in MSCS terms, large number of nodes.

7.2 Distributed Management

In distributed management software, two components are considered basic building blocks: a consistent view about which nodes are on-line, and the ability to communicate with these nodes in an all-or-nothing fashion [8].

The first building block is captured in a *membership service*: all nodes participate in a consensus algorithm to agree on the current set of nodes that are up and running. The system makes use of a failure detection mechanism that monitors heartbeat signals or actively polls other nodes in the system. The failure detector will signal the membership service whenever it suspects the failure of a node in the system. The membership service will react to this by triggering the execution of a distributed algorithm at all the nodes in the system, in which they agree upon which nodes have failed and which are still available. The joining of a new member in the system, does not require the nodes to run the agreement protocol, but can often be handled through a simple update mechanism.

The second fundamental component provides a special communication facility, with guarantees that exceed the properties provided by regular communication systems. Often in the process of managing a distributed system it is necessary to provide the same information to a set of nodes in the system. We can simplify the software design of many of the components on the receiving side of this information if we can guarantee that if one node receives this information, that all nodes will receive it. This *atomicity* guarantee allows nodes to act immediately upon reception of a message, without the need for additional synchronization. Often this atomicity guarantee is not sufficient for a system, as it does not only need be assured that all nodes will receive the update, but that all nodes will see the updates in the same order. This *total order* property makes the communication module a very powerful mechanism in the control of the distributed operation of the distributed system.

7.3 Practical Scalability

This chapter examines the membership and communication services of the Microsoft Cluster Service (MSCS) with an emphasis on their impact on the scalability of the system. MSCS, as shipped, officially supports 2 nodes, but in reality the software can be run on a 16-node NT server cluster. For the purpose of the scalability tests we extended the system software to run on 32 nodes.

Making systems scale in practice centers around the use of mechanisms to reduce the dependency of the algorithms on the number of nodes. In the past, two approaches have been successful in finding solutions to problems of scale: The first is to reduce the synchronous behavior of the system by designing messaging systems and protocols that allow high levels of concurrent operation, for example by de-coupling the sending of messages from the collecting of acknowledgements. The second approach is to reduce the overall complexity of the system. By building the system out of smaller (semi-)autonomous units and connecting these units through hierarchical methods, growing the overall system has no impact on the mechanism and protocols used to make the smaller units function correctly.

A third, more radical approach, which is under development at Cornell, makes use of gossip based dissemination algorithms. These techniques significantly reduce the number of messages and the amount of processing needed to reach a similar level of information sharing among the cluster nodes, while maintaining a superior level of robustness to various failures.

Given that cluster systems such as MSCS are used for enterprise computing, any instability of the system can have severe economic results. There is a continuous tradeoff between responsive failure handling and the cost of an erroneous suspicion. The system needs to detect and respond to failures in a very timely matter, but designers may choose a more conservative approach given the significant cost of an unnecessary reconfiguration of the system, caused by an incorrect failure suspicion. In general cluster server systems run compute and memory intensive enterprise applications and these systems experience a significant load at times, reducing the overall responsiveness. Scaling failure detection needs intelligent mechanisms for fault investigation [81,99] and requires the failure detectors to be able to learn and adapt to changes [80].

7.4 Scalability goals of MSCS

The Microsoft Cluster Service is designed to support two nodes, with a potential to scale to more nodes, but in a very limited way. MSCS successfully addresses the needs of these smaller clusters. The cluster management tools are a significant improvement over the current practice and they are a major contribution to the usability of clusters overall.

The research reported here is concerned with scaling MSCS to larger numbers of nodes (16 - 64, or higher), which is outside of the scope of the initial MSCS design. There are three areas of interest:

1. Can the distributed algorithms used in MSCS be a solid foundation for scalable clusters?
2. Are there any architectural bottlenecks that should be addressed if MSCS needs to be scalable?

3. If MSCS is extended with development support for cluster aware applications are the current distributed services a good basis for these tools?

This work should not be seen as criticism of the current MSCS design. Within the goals set for MSCS it functions correctly and will scale to numbers larger then originally targeted by the cluster design team.

7.5 Cluster Management

The algorithms used in MSCS for membership and total ordered messaging are a direct derivative of those developed in the early eighties for Tandem as used in the NonStop systems [16,53]. Nodes in a Tandem system communicated via pairs of proprietary inter-processor busses, which, in 1985, provided a 100 Mbit/second transfer rate. Parts of the messaging side of the algorithms was implemented in interrupt handlers to provide minimal system overhead. It is also important to notice that the bus provided a broadcast facility.

Although MSCS has a kernel module that implements some of the messaging and failure detection, the membership and global update algorithms are implemented in an NT service, the Cluster Service, which runs at user level. The *Cluster Service* holds in total 11 managers, each responsible for a different part of the cluster service functionality. Next to the membership and communication managers, there are managers for resource and failover management, for logging and checkpointing, and for configuration and network management.

In the following sections three of the components are examined in detail: first the kernel module which holds the cluster communication and failure detection functionality. Secondly the join process and the failure reconfiguration of the membership module are analyzed. The last analysis is that of the global update communication module.

7.6 Cluster Network

MSCS provides a kernel based cluster network interface, *ClusNet*, which presents a uniform interface to networks available for intra-cluster communication. ClusNet supports basic datagram communication to each of the nodes, using an addressing scheme based on simple node identifiers which are assigned to nodes when they are first configured for use in the cluster. To support reliable communication ClusNet provides a transport interface used by MS-RPC, which is directly derived from DCE RPC.

ClusNet is capable of managing a redundant networking infrastructure, automatically adapting packet routing in case of network failure.

7.6.1 Node Failure Detection

MSCS implements its Failure Detection (FD) mechanism using heartbeats. Periodically every node sends a sequenced message to every other node in the cluster, over the networks that are marked for internal communication. Whenever a node detects a number of consecutive missing heartbeats from another node it sends an event to the cluster service which uses this event to activate the membership reconfiguration module.

In the current MSCS configuration heartbeats are sent every 1.2 seconds and the detection period for a node suspicion is 7.2 seconds (6 missed heartbeats). The timing values are not adaptive.

The cluster network module does not exploit any broadcast or multicast functionality, and thus each heartbeat results in (*number_of_nodes-1*) point-to-point datagrams. In our test setup of 32 nodes, the cluster background traffic related to heartbeats is 800 messages per second. With 32 nodes active and an otherwise idle network the mechanism works flawless and the packet loss observed was minimal. Tests which replaced the Fast-Ethernet switches with hubs showed that the packet trains sometimes caused significant Ethernet-level collisions on the shared medium. Adding processing load to the systems resulted in variations in the inter-transmission periods. False suspicions were never seen.

When adding processing load and additional load on the network frequently single heartbeat misses were observed, but the timing values for generating a failure suspicion events are set very conservative, and as such these missed heartbeats never generate any false suspicions. The failure detection times could be set more aggresively based on the network capabilities but field tests with varying heartbeat configurations have shown that legacy hardware limitations warrant this conservative approach.

7.7 Node Membership

The MSCS membership manager is designed into two separate functional modules: the first handles the joining of nodes and a second, regroup, implements the consensus algorithm that runs in case of a node failure.

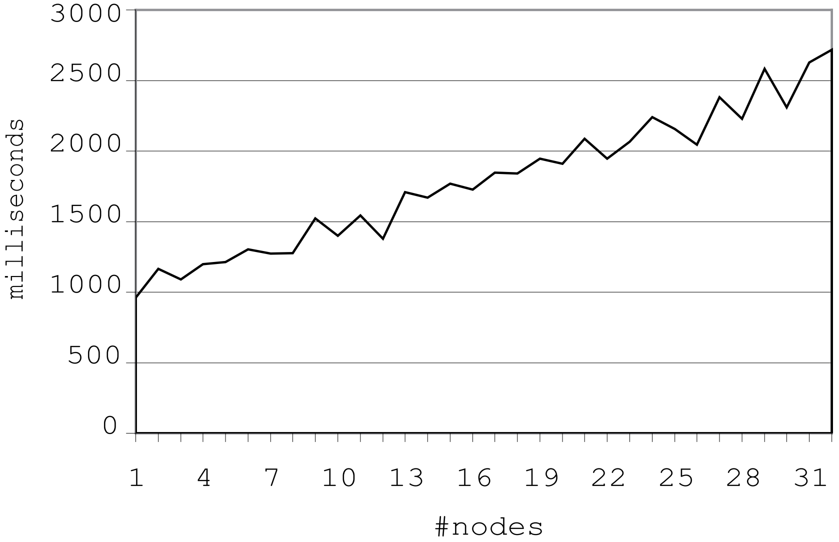


Figure 7.1. Join Latency under ideal conditions

7.7.1 Join

The join algorithm starts with a discovery phase in which the joining node attempts to find other nodes in the cluster. If this fails the node tries to form a cluster by itself, the details of the cluster-form operation can be found in chapter 5. After the node has discovered which cluster nodes are currently running it selects one of the nodes and petitions for membership of the cluster. The selected node, dubbed the sponsor, announces the new node to all active cluster members, transfers all the up-to-date cluster configuration to the new node, and waits for the node to become active. The different phases of the join and their distributed complexity are described in detail in the following paragraphs

Phase 1: **Discovery**. When a cluster service starts it attempts to connect to each of the other known nodes in the cluster, using RPC over a regular UDP transport. This sponsor discovery mechanism has a high degree of concurrency: a thread is started for each connection probe. The joiner waits for all threads to terminate, which occurs when a connection is established or the RPC binding operation fails after a time-out. As the joiner waits for all threads to terminate, the delay the joining node experiences is based on the time-out period of a RPC connection to a single node that is not up. The timeout value for RPC out-of-the-box is approximately 30 seconds, but it can be manipulated to reduce the discovery phase to 10 seconds.

In all observed cases, the joining node always selected the holder of the cluster IP address to sponsor its join. The cluster IP address is a single address that migrates to a node that functions as the access point for administrative purposes: if the cluster is running there is *always* a node that holds this IP address. By modifying the startup phase to start by attempting to connect to this address first before probing all the other nodes, it is possible to reduce this phase of the join process to under a second. This approach also avoids starting a number of threads that is equal to the number of nodes in the cluster.

Phase 2: **Lock**. From the nodes that are up, the joiner selects one node to *sponsor* its membership in the cluster. The first action by the sponsor is to acquire a distributed global lock to ensure that only a single join is in progress. Acquiring of the lock is performed using a global update (GUP) method.

The use of GUP makes this phase dependend on the number of active nodes. Details on the performance and scalability of GUP can be found in section 6.7.

Phase 3: **Enable Network**: Using a sequence of 5 RPC calls to the sponsor, the joiner retrieves all information on current nodes, networks and network interfaces. Following this the joiner performs an RPC to each active node in the cluster for each network interface a node is listening on, and the contacted node in return performs an RPC to the joiner to enable symmetric network channels. After this sequence the node security contexts are established which again requires the joining node to contact all other active nodes in the cluster, in sequence.

This phase depends on the number of active nodes in the cluster. An unloaded 31 nodes cluster, on average, performs this sequence of RPC's in 2-4 seconds. On a moderately loaded cluster, frequently this phase takes longer than 60 seconds, causing the join operation to time-out at the sponsor, resulting in an abort of the join.

Phase 4: **Petition for Membership**: The joiner requests the sponsor to insert the node into the membership. This is a 5-step process directed by the sponsor.

1. The sponsor broadcasts to all current members the identification the joining node.
2. The sponsor sends the membership algorithm configuration data to the joiner
3. The sponsor waits for the first heartbeat from the new joiner.
4. The sponsor broadcasts to all current members that the node is alive
5. The sponsor notifies the joiner that it is inserted in the membership

The broadcasts are implemented as series of RPC calls, one to each active node in the cluster. On an unloaded cluster and network the serialized invocation of RPC to 30 nodes takes between 100 and 150 milliseconds. When loading the systems with compute and IO tasks, the RPC times vary widely from 3 millisecond to 3 second per RPC. Broadcast rounds to all 30 nodes were observed taking more than 20 seconds to complete (with exceptions up to 1 minute). As this phase is under control of the sponsor the join is not aborted because of a time-out. It can abort on a communication failure with any of the nodes.

In step 3 the detection of the new heartbeat is delegated to ClusNet, which performs checks every 600 millisecond, resulting in an average waiting period between 0.6 and 1.2 seconds

Phase 5: **Database synchronization**. The joiner synchronizes its configuration database with the sponsor. In the experimental setup this database was of minimal size and never out-of-date. As the retrieving of the database updates is not dependent on cluster size, no further tests were performed on this phase.

Phase 6: **Unlock**. The newly joined node uses its access to the global update mechanism to broadcast to all nodes that it now is full operation and that the global lock should be released.

The join operation is very much dependent on the number of nodes in the system. Figure 7.1 shows the times for a join under optimal conditions. All RPC calls in the algorithms are serialized and at minimum there are (10 + 7 * number_of_nodes) calls. Joining the 32nd node to the cluster requires at least 227 sequential RPC's. This approach collapses under load, frequently it is impossible to join any nodes if only a moderate load is placed on the nodes and the system has more than 10-12 nodes.

7.7.2 Regroup

Upon the receipt of a node failure event generated by ClusNet the Cluster Service starts the reconfiguration algorithm, dubbed *regroup*. The algorithm runs in 5 phases, with the transition to each new phase determined after it is believed that all other nodes have finished this phase, or when, in the first two phases, timers expire.

During regroup the nodes periodically (300ms) broadcast their current state information to all other nodes using unreliable datagrams. The state is a collection of bitmasks, one for each phase, describing whether a node has indicated it has passed

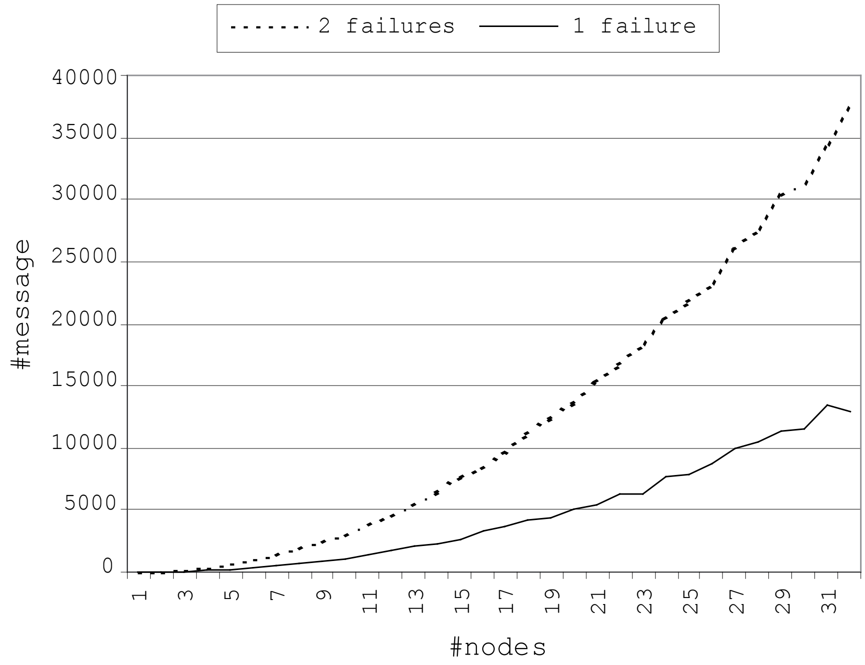


Figure 7.2. Number of messages in the system during regroup

a phase. It is not necessary for each node to have heard for each other node in a phase; information about which other nodes a certain node has heard of is shared. For example if node 1 indicates that it has received a regroup message from node 2, node 3 uses this without that it actually needs to receive a message from node 2 in that phase. Also included in the state is a connectivity matrix in which nodes record whether they have seen messages from the other nodes and what connectivity information has been recorded by the other nodes.

The 5 phases of the regroup algorithm are the following:

Phase 1: **Activate**. Each node waits for a local clock tick to occur so that it knows that its timeout system can be trusted. After that it starts sending and collecting status messages. It advances to the next stage if

1. All current members have been detected to be active (e.g. there was a false suspicion),
2. If there is one single failure and a minimal time-out has passed or,
3. When the maximum waiting time has elapsed and several members have not yet responded.

The minimum timeout for phase 1 is 2.4 second, if all but one node have responded in this time period it is assumed that there was a single failure and the algorithm moves to the next phase. If multiple nodes do not respond, the algorithm waits for 9.6 seconds to move to the next phase. If for some reason the regroup algorithm times out in a different phase or when there are cascading starts of the regroup algorithm at several nodes, the algorithm executes in *cautious* mode and always waits for the maximum timeout to expire.

Phase 2: **Closing**. This stage determines whether partitions exist and whether the current node is in a partition that should survive. The rules for surviving are:

1. The current membership contains more than half the original membership.
2. Or, the current membership has exactly half the original members, and there are at least two members in the current membership and this membership contains the tie breaker node that was selected when the cluster was formed.
3. Or, the original membership contained exactly two members and the new membership only has one member and this node has access to the quorum resource.

After this the new members select a tie breaker node to use in the next regroup execution. This tiebreaker then checks the connectivity information to ensure that the surviving group is fully connected. If not it prunes those members that do not have full connectivity. It records this pruning information in its regroup state, which is broadcast to all other nodes. All move to stage 3 upon receipt of this information.

In case of incomplete connectivity information the tiebreaker waits for an additional second to allow all nodes to respond.

Phase 3: **Pruning**. All nodes that have been pruned because of lack of connectivity halt in this phase. All others move forward to the first cleanup phase once they have detected that all nodes have received the pruning decision (e.g. they are in phase 3).

Phase 4: **Cleanup Phase One**. All surviving nodes install the new membership, mark the nodes that did not survive the membership change as down, and inform the cluster network to filter out messages from these nodes. Each node's Event Manger then invokes local callback handlers to notify other managers of the failure of nodes.

Phase 5: **Phase Two**. Once all members have indicated that the Cleanup Phase One has been successfully executed, a second cleanup callback is invoked to allow a

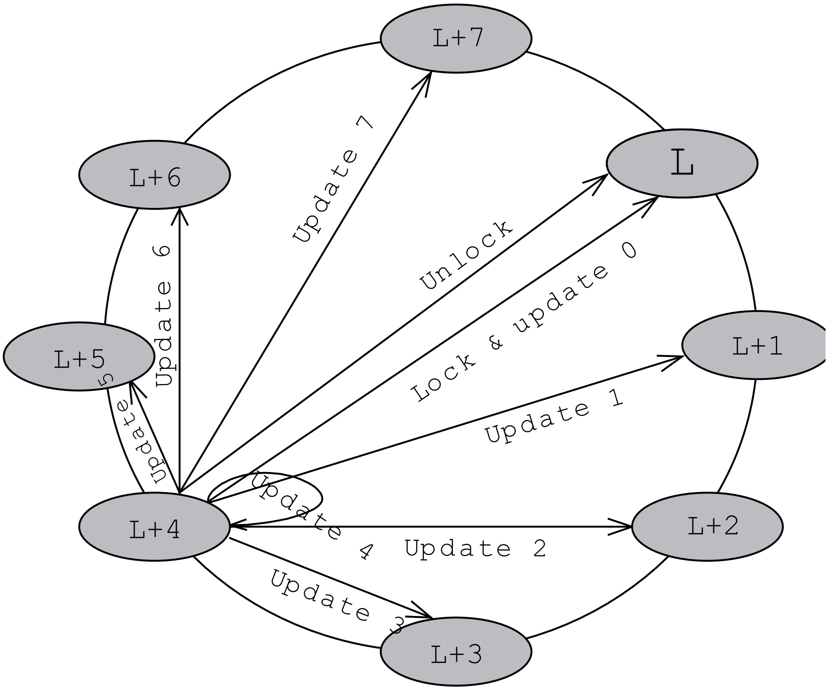


Figure 7.3. Global Update Sequence

coordinated two-phase cleanup. Once all members have signaled the completion of this last cleanup phase they move to the regular operational state and seize the sending of regroup state messages.

The regroup algorithm in its first two phases is timer driven and the algorithm makes progress independent of the number of nodes in the cluster. The transitions of the next 3 phases are dependent on the number of nodes in the system, but the "information sharing" mechanism makes the system robust in dealing with sporadic message loss.

The state information is broadcast by sending point-to-point datagrams to each node in the cluster. With an inter-transmission period of 300 millisecond, and 31 nodes in the cluster, this generates a background traffic of over 3000 messages/second. A single failure reconfiguration has an average runtime of 3 seconds and thus generates around 10,000 messages. A two-node failure, with a full running cluster is likely to generate between 30,000 and 40,000 messages. Figure 7.2 details the observed messages in the system during regroup.

7.8 Global Update Protocol

It is essential for a distributed management system to have access to a primitive that allows consistent state updates at all nodes. MSCS uses the Global Update Protocol (GUP) for this purpose. Although the protocol is described as providing atomicity, its implementation has the stronger property of providing total ordering to its update messages.

When a node starts an global update operation, it first competes for a transmission lock managed by a node that is assigned the functionality of the locker node. Only one transmission can be in progress at a time. If the sender can not obtain the lock it is queued on the lock waiting list and blocks until it reaches the head of the queue. With the lock request the sender also transmits its update information to the locker node which applies it locally, and stores the message for later replay under certain failure scenarios. While holding the lock the sender transmits its update to all other active nodes in the cluster and terminates the transmission with a final message to the locker node which releases the lock (see figure 7.3).

To transmit the messages to all other nodes, the sender organizes the cluster nodes into a circular list, ordered by NodeId. After it acquired the lock, the sender sends its updates starting with the node that is after the locker node in the list. The sender works through the list in order, wrapping when it reaches the last node in the cluster to the first node and stops when it once again reaches the locker node. The transmission is finished with an unlock message to the locker node.

Acquiring the lock before performing the updates guarantees that only one update is in progress at a given time, which gives the protocol the total ordering property. Atomicity (if one surviving node applies the update, all other surviving nodes will) is achieved through the implementation of a number of fault-handling scenarios.

1. *The sender fails*: the locker node takes over the transmission and completes it.
2. *A receiver fails*: wait for the regroup to finish and then finish the transmission.
3. *The locker node fails*: the next node in the node list is assigned locker functionality and the sender treats it as such.
4. *The sender and locker fail*: if the node following the locker has received the update already, in its role as new locker it takes over the transmission.

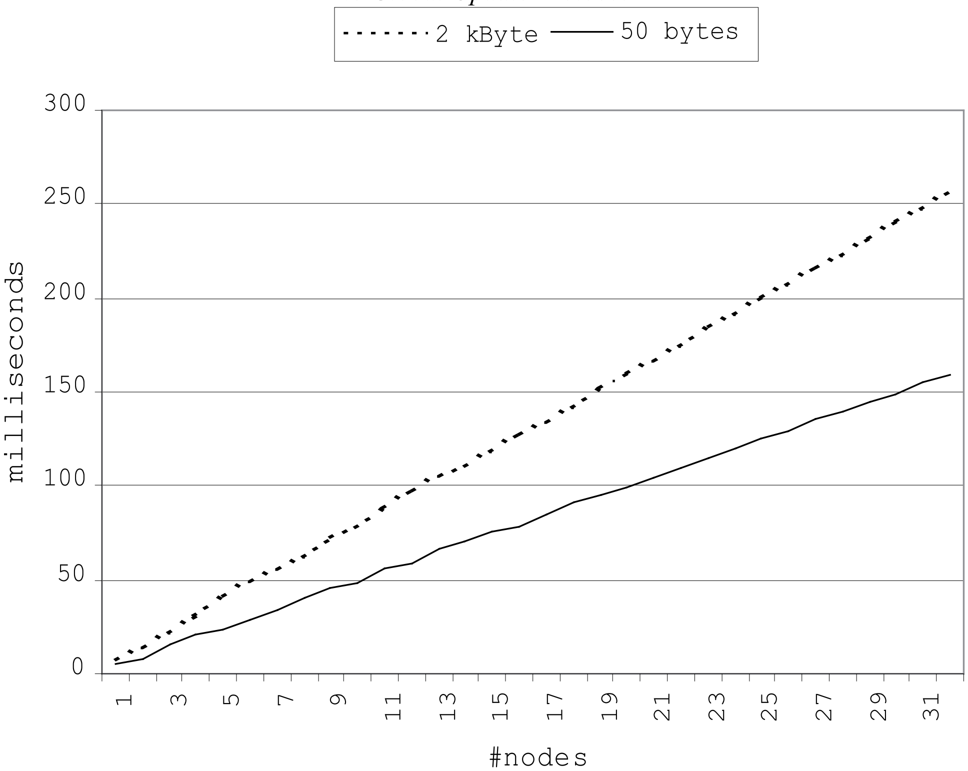


Figure 7.4. Latency of GUP under ideal conditions.

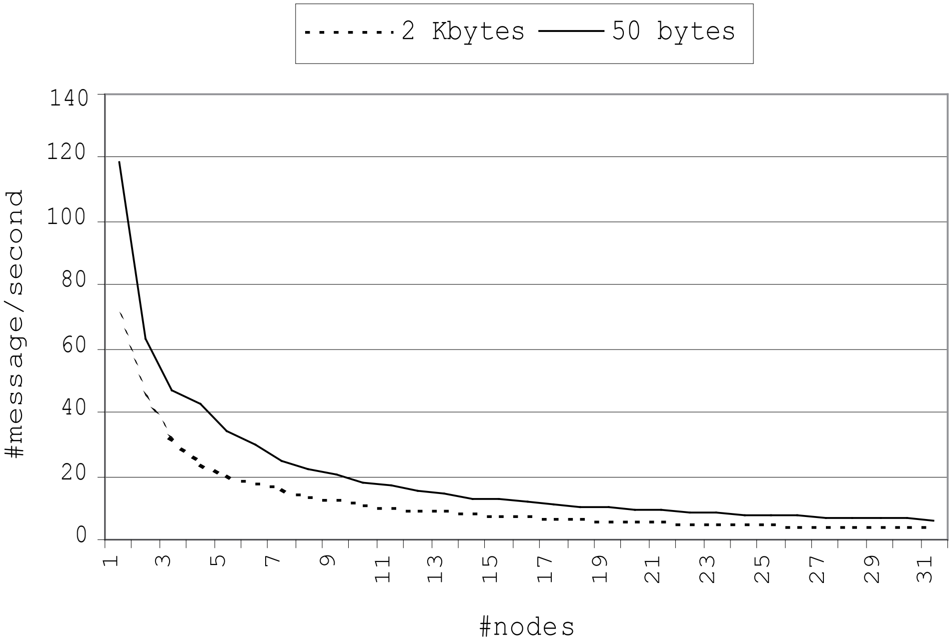


Figure 7.5. GUP Throughput under ideal conditions

5. *All nodes that received an update and the sender fail:* pretend the update never happened.

The protocol is implemented as a series of RPC invocations. If an RPC fails, the sender waits for the regroup algorithm to run and install a new membership. GUP will then finish the update series based on the new membership.

Given the strict serial execution of the protocol, its performance is strongly dependent on the number of nodes in the system. The implementation enforces no time bound on the execution of an RPC and any node can introduce unbounded delays as long as RPC keep-alives are being honored.

Repeated measurements show huge variations in results, with the variations being amplified as the number of nodes increases. When a moderate load is placed on the nodes it becomes impossible to produce stable results. These variations can be contributed to the RPC trains, which repeatedly transfer control to the operating system while blocking for the reply. Upon arrival of the reply at the OS level, the Cluster Service needs to compete with other applications that are engaged in IO, to regain CPU control. The non-determinism of the current load state of the system introduces the variances.

The latency of the protocol in an ideal setting is shown in figure 7.4, the message throughput in figure 7.5. With 32 nodes the system can handle 6 small (50 bytes) updates/second or 4 larger (2 Kbytes) updates/second.

With systems under a load the protocol breaks down with more then 12 nodes in the cluster. With 10 nodes frequently transmissions are observed that take 2-5 seconds to complete. With 32 nodes transmission times up to one minute were recorded.

7.9 Discussion

When evaluating the scalability of the distributed components of MSCS it is necessary to separate two issues: the algorithms used and their particular implementation.

7.9.1 Failure Detection

MSCS is willing to tolerate a long period of silence (7 seconds) before a failure suspicion is raised. This allows for the implementation of mechanisms that can easily deal with large number of nodes. The important scale factor is the number of messages that the nodes need to process both at the sending and the receiving side. Implementing the heartbeat broadcast using repeated point-to-point datagrams does not introduce any problems with 32 nodes, but there is a clear processing penalty at the sender and it will limit the growth to larger numbers.

In an unstructured heartbeat scheme (every node sends heartbeats to all other nodes), the load on the sender and on the network can be significantly reduced by using a true multicast primitive for disseminating the heartbeats. It also removes the sender's dependency on the number of nodes in the system. However, the number of messages a receiver has to process remains proportional to the number of nodes in the system.

More structured approaches have been proposed to reduce the overall complexity of failure detection by imposing a certain structure on the cluster, and localizing failure detection within that structure. A popular approach is to organize the cluster nodes in a logical ring [3,65] where nodes only monitor neighbor nodes in the ring and a token rotates through the ring to disseminate status information. In this scheme however, the token rotation time is dependent on the number of nodes, and the scheme thus has clear scalability limits.

Another aspect of scaling failure detection is the increased chance of multiple concurrent node failures in the cluster. The MSCS mechanism handles multiple failures just as efficient as single failures, while most of the structured failure detection schemes have problems with timely detection of multiple failures and fast reconfiguration of the imposed structure.

The most promising work on failure detection for larger systems is the use of gossip and other epidemic techniques to disseminate availability information [81]. These detectors monitor hundreds of nodes while still providing timely and accurate detection, without imposing any significant increased load on nodes and networks.

7.9.2 Membership Join

The observation that it frequently was impossible to join the 15th or higher node into the cluster is an artifact of the fact that MSCS was not implemented with a large number of nodes in mind. The join reject happens in the phase that is not under control of the sponsor node and where the new node is setting up a mesh of RPC bindings and security contexts with all other active nodes. With 32 nodes this phase is close to a 100 RPCs and any load on the nodes causes significant variations in these serialized executions.

There is no fundamental solution to the problem; if the RPC infrastructure needs to be maintained, the setup phase is needed and some tolerance is needed to allow the mesh to be established. A possible solution would for the joiner to update the sponsor on its progress in this phase to avoid a join rejection.

7.9.3 Membership Regroup

The membership reconfiguration algorithm works correct under all tested circumstances, independent of the number of nodes used. There are two mechanisms that ensure that the operation performs well, even with a larger number of nodes: (1) The operation is fully distributed, the constant broadcasting of state allows nodes to rely solely on local observation of global state. (2) The sharing of "*I-have-heard-from-node-X*" information among nodes, makes that the nodes can move to the next phase without having received status messages from all nodes.

Given that a node failure suspicion is not raised until 7 seconds of silence by a node and the first phase of regroup waits for an additional 3 seconds, a problematic node has 10 seconds to recover from some transient failure state. As no false suspicions were ever observed, the timeouts in the first two phases of regroup can be considered to be very conservative. In all observed cases the current membership state was already established well within a second, the remaining time (2-9 seconds) was spent waiting for the failed nodes to respond. As the first phase is dominant in the execution time of the whole regroup operation, a reduction in time can be achieved by combining the failure detection information with the observed regroup state.

A major concern in scaling the regroup operation is the number of messages exchanged. A typical run with 32 nodes generates between 10,000 and 40,000 messages. The status message broadcasts are implemented as series of point-to-point datagrams, which has two major effects: (1) the number of messages generated for the regroup operation grows exponential with the number of nodes and (2) the transmission of 32 identical messages every 300 milliseconds introduces a significant processing overhead at the sender. The regroup algorithm is run at the cluster service, which introduces a user-space/kernel transition for each message, with associated overhead. Introduction of a multicast primitive will allow the implementation to scale at least linearly with the number of nodes and would remove the processing overhead from the sender of status messages. A second needed improvement is the implementation of the memebrship engine at kernel level, allowing for lower system overhead and a more predictable execution of the algorithms.

7.9.4 Global Update Protocol

The absence of any concurrency in the message transmission in GUP causes a strict linear increase in latency and decrease in throughput when the number of nodes in the cluster grows.

The serialized and synchronous nature of the protocol is amplified in the particular MSCS implementation. The protocol was originally developed for updating shared OS data-structures, with the update routines running in device interrupt handlers. In MSCS the protocol is implemented using a series of RPC calls to user-level services. This change in execution environment exposes the vulnerability of the strict serialized operation.

There is no quick solution for the problems that this GUP implementation presents us with. To emulate the original Tandem execution environment the complete Cluster Service would need to be implemented as a kernel service, which at this point seems impractical.

Replacing GUP with a protocol that provides the same properties but exhibits a more scalable execution style seems preferable. This introduces a number of other complexities, for example many of the currently popular total ordering protocols rely on a tight integration of membership and communication to ensure correct failure handling. This would result in replacing *regroup* as well as GUP.

7.10 Conclusions

In this research some of the scalability aspects of the Microsoft Cluster Service were examined. When revisiting the three questions from section 6.3 the following is concluded:

Can the currently used distributed algorithms be a solid foundation for scalable clusters?

Both failure detection and regroup scale to the numbers that were tested in the experiments. When scaling to larger numbers the state processing at receivers will become an issue. The serialized nature of GUP limits its scalability to 10-16 nodes in the current MSCS setup.

Are there any architectural bottlenecks that should be addressed if MSCS needs to be scalable?

The major issue in both failure detection and regroup is the implementation of a broadcast facility using repeated point-to-point messages. This introduces a significant overhead on the sender and on the network, and needs to be replaced by a simple multicast primitive. The RPC trains in the membership join operation and in GUP, create a major obstacle for scalability, especially when the systems operate under a significant load.

If MSCS is extended with development support for cluster aware applications are the current distributed services a good basis for these tools?

Support for cluster aware applications has strong requirements in the area of application and component management and failure handling, and requires efficient communication and coordination services. These services would need to be implemented using GUP, which is, in its current form, unsuitable to provide such a service.

To support cluster aware applications a better integration of membership and communication is needed. This will allow for the implementation of a very efficient communication service with properties similar to GUP. Such a service is capable of providing a solid basis for application and component level management and failure handling, and will offer efficient communication and coordination services.

Chapter 8

The Galaxy Framework for the Scalable Managament of Enterprise-Critical Cluster Computing

In this chapter we present the main concepts behind the Galaxy cluster management framework. Galaxy is focused on servicing large-scale enterprise clusters through the use of novel, highly scalable communication and management techniques. Galaxy is a flexible framework built upon the notion of low-level management of cluster farms, where within these farms islands of specific cluster management types can be created. A number of cluster profiles, which describe the components used in the different cluster types, are presented, as well as the components used in the management of these clusters.

8.1 Introduction

The face of enterprise cluster computing is changing dramatically. Whereas in the past clusters were dedicated resources for supporting particular styles of computing (OLTP, large batch processing, parallel computing, high-availability)[27,49,58,70, 88,97], modern Data Centers hold large collections of clusters where resources can be shared among the different clusters or at least easily re-assigned to hotspots within the overall Data Center organization. A Data Center may see a wide variety of cluster types; cloned services for high-performance document retrieval, dynamic partitioning for application controlled load balancing, hot-standby support for business-critical legacy applications, parallel computing for autonomous data-mining and real-time services for collaboration support.

It must be obvious that such a complex organization with a variety of cluster types exceeds the scope of conventional cluster management systems. In the Galaxy project we are concerned with constructing a framework for data-center-wide cluster management. Using this framework a management organization can be constructed that controls clusters and cluster resources in an integrated manner, allowing for a unified, data-center-wide approach to cluster management.

Galaxy provides a multi-level approach. In its most basic configuration this offers two abstractions; the first is the *farm*, which is the collection of all nodes that are managed as a single organization, and the second is a *cluster* which is an island of nodes within the farm that provides a certain cluster style management, based on a *cluster profile*. A profile is a description of a set of components, implementing advanced distributed services that make up the support and control services for that particular cluster.

Extreme care has been given to issues related to scalability, especially with respect to the components implementing the communication and distributed control algorithms. The novel techniques used in the communication, membership and failure detection modules allow scalability up to thousands of nodes. In our limited experimental setup we have shown that scaling up to 300-400 nodes is possible, giving us confidence that our future experiments with even larger sets of nodes will continue to show excellent scalability.

The first phase of the development of Galaxy is complete; the framework and a set of generic management components are implemented and a number of example cluster profiles and their specialized management components are in daily use. The system is developed for the Microsoft's Windows 2000 operating system, integrating tightly with the naming, directory, security, and other distributed services offered by the operating system. Galaxy has been selected by a major operating system vendor as the basis for its next generation cluster technology.

This chapter is organized as follows: in sections 8.2 and 8.3 the model underlying the framework is presented and in sections 8.4 and 8.5 we provide some background on our approach to building scalable distributed components. In sections 8.6 through 8.9 details are given on the design of the farm and cluster support. The chapter closes with some references to related work, a description of our upcoming distribution, and plans for the immediate future.

8.2 General Model

Galaxy uses a multi-level model to deliver the cluster management functionality. The basic abstraction is that of a *Farm* [27], which is a collection of managed machines in a single geographical location. The farm can consist of potentially thousands of nodes, and is heterogeneous in nature, both with respect to machine architectures and network facilities. Galaxy provides a set of core components at each node in the farm that implement the basic control functionally.

The administrative personnel responsible for the overall operation of the farm has a set of farm management tools, within which the *Cluster Designer* is most prominent. This tool allows an administrator to construct islands of *clusters* within the farm, grouping nodes in the farm together in tighter organizations to perform assigned functionality. Different cluster styles are defined using *Cluster Profiles*, which describe the set of components that make up the management and application support functionality for that particular cluster style. A new cluster is created based on a profile, and when a node is added to the cluster it will automatically instantiate the components described in the profile for the role this particular node is to play, and join the cluster management group for purpose of intra-cluster failure monitoring and membership management.

Both at the farm and cluster level, the system architecture at a node is identical. At the core one finds a management component that implements membership, failure detection and communication. This management service is surrounded with a set of service components implementing additional functionality, such as an event service, a distributed process manager, a load management service, a synchronization service and others. These components export a public service specific API, to be used by other management services or applications. Internally each of these components implements a standard service API so that the components can be managed by the farm or cluster management service. The components receive service specific membership information, indicating the activity status of this service at the other nodes in the cluster or the farm.

At a node that is part of a cluster, one would see two management services: one for the farm and one for the cluster, each with its own set of managed service components. To correctly visualize the hierarchy one should imagine an instance of the Cluster management service as a component managed by the Farm management server. In principle a node can be part of multiple clusters, although we have only found one case in which overlapping was practical. We believe however that if creation of clusters as lightweight management entities is a simple enough action, we may see scenarios in which overlapping does play an important role.

The existing case where overlapping clusters is used in Galaxy arises when grouping a collection of nodes into a management cluster. This cluster runs a set of services that is specific for controlling the overall management of the farm and configured clusters. The nodes in this cluster can be dedicated management nodes or can be part of other clusters, performing some general management tasks as background activities. The services that are specific for the management cluster are mainly related to event collection, persistency of overall management information and the automatic processing of management events.

8.2.1 Extensions to the basic model

After Galaxy had been in use in production settings the need arose for two additional management abstractions that were not targeted in the initial design of Galaxy. The first extension, the notion of a *meta-cluster*, was introduced to deal with the limited scalability of legacy cluster applications. The second extension, *geoplex management*, addresses the need to have cluster management structures span nodes in multiple farms, where the farms are geographically distributed.

Even though the distributed systems technology underlying Galaxy is highly scalable, the technology on which legacy applications are based often is limited in scale. Ideally an application cluster would consist of all the nodes necessary to implement the functionality, but the limited scalability forces the cluster designer to break the cluster into a set of smaller clusters based on the maximum scale of the cluster application. To still be able to consistently manage the resources shared among these smaller clusters, Galaxy implements a notion of a *meta-cluster* which provides meta-cluster wide membership and communication primitives.

To manage clusters in a *geoplex* setting, Galaxy was augmented with a technology similar to the meta-clusters, but which functions at the farm level instead of the cluster level. In a *geoplex* membership entities are farms, and failure-detection and membership tracking is performed at the level of complete farms. Through geoplex-wide communication primitives the farms share node and cluster level membership information.

Galaxy does not allow clusters to span nodes in multiple farms; however it does allow meta-clusters to encapsulate clusters that are in different farms to provide management of functionality offered through geographical distribution.

8.3 Distribution model

Distribution support for clusters comes in two forms: first there are the distribution services to support the farm and cluster management infrastructure itself, these services provide the core mechanisms for all the distributed operations in Galaxy. Secondly there is the support offered to the applications that run on the clusters. In Galaxy the latter is offered through the services that are unique for each cluster configuration.

A very natural approach in structuring these services is to model them as groups of collaborating components [6,7]. This notion of groups naturally appears in many places in the system, whether it is in naming, where one wants to be able to address the complete set of components implementing a service, in communication where one wants to send messages to all instances of a service, or in execution control where you want to synchronize all service instances, etc. The notion of a group appears from the core communication level all the way to the highest abstraction level where one wants the farm and contained clusters to be viewed as groups, to access and address them as single entities.

In Galaxy the *process group* model is used throughout the whole framework, both in terms of conceptual modeling of the system, as in the technology used to provide the distribution support. In a modern architecture such as Galaxy, groups no longer necessarily consist of processes but are better viewed as interacting components. The technology that allows us to design the distributed components using a group abstraction provides at its core a view of the components in the group through a *membership service*, identifying which instances are presently collaborating. The service uses a *failure detector* to track the members of the group and to notify members of changes in the membership. The membership also provides a naming mechanism for communication, both for communication with the complete group as well as with individual members [8]. Modeling collaboration components as a group allows us to build support for complex interaction patterns where, for example message atomicity, request ordering or consensus on joint actions, play an important role. This core functionality is used to implement the various distributed operations such as state sharing, synchronization, quorum based operations, etc.

The communication and membership technology used to provide the group abstraction as part of Galaxy is a crucial component in achieving scalability. In Galaxy a new group communication system is used, where most of the limits on scalability that these systems exhibited in the past have been overcome.

8.4 Scalability

Cornell's Reliable Distributed Computing group has been building communication support for advanced distributed systems for the past 15 years, resulting in systems such as the Isis Toolkit and the Horus and Ensemble communication [8,80] frameworks. Most recently, the Spinglass project focuses on issues of scalability in distributed systems, with early results in the form of highly scalable communication protocols [10], failure detection [81] and resource management [82]. The scalability vision that drives the research in Spinglass, also has driven the design and development of Galaxy.

Building on our experiences of building distributed systems, mainly in the industrial sector, we have collected a set of lessons that are crucial to developing scalable systems. These lessons drive our current research and are applied in the design of Galaxy project. The five most important are:

1. *Turn scale to your advantage*. When developing algorithms and protocols you have to exploit techniques that work better when the system grows in scale. Adding nodes to a system must result in more stable overall system and that provides more robust performance instead of less. Any set of algorithms that cannot exploit scaling properties when a system is scaled up is likely be the main bottleneck under realistic load and scaling conditions. Successful examples of this approach can be found in the gossip-based failure detector work [81], the bimodal multicast protocols [10], and the new multicast buffering techniques [72].

2. *Make progress under all circumstances*. Whenever a system grows it is likely that, at times, there will be components that are experiencing performance degradations, transiently or permanently, or even may have permanently failed or brought off-line. Traditionally designed cluster management systems all experience an overall performance degradation whenever certain components fail to meet basic performance criteria, often resulting in a system that runs at the pace of its slowest participant. These dependencies need to be avoided at the level of the cluster management infrastructure to avoid scenarios in which bad nodes

in the system drag down the complete cluster or, at the worst case, the farm [11]. Systems built based on the epidemic technology provide a probabilistic window within which the system is willing to tolerate failing components, without affecting the overall system. If, after this time window, the component has not recovered it is moved from the active set to the recovery set, where system dependent mechanisms are deployed to allow the component to catch up. This minimizes the impact on the overall system performance [8].

3. *Avoid server-side transparency*. If applications and support systems are designed in a manner unaware of the distributed nature of the execution environment, they are likely to exhibit limited scalability. Achieving true transparency for complex, production quality distributed systems is close to impossible, as the past has shown. Only by designing systems explicitly for distributed operation, will they be able to handle the special conditions they will encounter and thus be able to exhibit true scalability (see chapter 7).

4. *Don't try to solve all the problems in the middleware*. As a consequence of a misguided attempt to guarantee transparency, research systems have focused providing support for all possible error conditions in the support software, attempting to solve these without involving the applications they need to support. The failure of this approach demonstrates a need for a tight interaction between application and support system. For example, applications can respond intelligently to complex conditions such as network partitioning and partition repair [103].

5. *Exploit intelligent, non-portable runtimes.* Cross platform portability is a laudable goal, but to build high performance distributed management systems one needs to resort to construct modules that encapsulate environment specific knowledge. For example the application must be able to inspect the environment using all available technology, and use the resulting knowledge in an intelligent manner. Any system that limits itself to technologies guaranteed to be available cross-platform, condemns itself to highly inefficient management systems, that are unable to exploit platform specific information which is necessary in achieving high-performance.

   An important example is failure detection of services, hosts and networks. Approaching this problem in a cross-platform, portable manner is likely to result in a system that can only exploit a heartbeat mechanism, possibly augmented with some SNMP status information [99]. Using specific knowledge about the environment will allow for construction of highly efficient failure detectors. A simple example is that some systems make ICMP error information available to the sender of the UDP message that triggered the error response, often an indication that the recipient is no longer available, and the failure detection can be cut short. Another example arises in the case where nodes are connected through a VIA interconnect, which provides ultra-reliable status information about the interconnected nodes, providing failure detection without the need for nodes to communicate.

8.5 Distribution support system

The Galaxy framework is designed with the assumption that there is access to a high-performance communication package, which provides advanced, configurable distributed systems services in style of the Horus and Ensemble systems. These services have to exhibit the scalability properties needed to provide a solid base for a scalable management system, as described in the previous section. In the current implementation we use an advanced communication package based on the new Spinglass protocols.

The *Scalable Group Communication Service* (SGCS) package is similar to the Horus and Ensemble in that it is built around a set of configurable communication stacks, which implement a unified mechanism for point-to-point and group communication, and provides an abstraction for participant membership and failure detection. This new system represents an advancement over these previous research systems in that it is based on the scalable failure detection, communication and state sharing protocols developed in the Spinglass project, and uses a next generation stack construction mechanism, allowing for asymmetric, optimized send and receive paths, and a message management paradigm that provides integration with user-level network facilities.

The communication system provides mechanisms for the creation of communication of multi-party communication channels with a number of reliability and ordering guarantees, and offers a unified name space for group and point-to-point communication.

The protocol core of the system is portable but its runtime is specifically adapted to function well in kernel environments, and parts of the system used in this research runs as a kernel module in Windows 2000. The complete package is encapsulated as a component server integrating well with existing development tools.

8.5.1 Failure detection and membership services

Essential for a reliable distributed system is the knowledge of which participants are available and which are off-line or have failed. To achieve this most systems deploy a failure detection service that tracks nodes and reports possible failures to subscribers. In Galaxy a multi-level failure detection service is used that can be configured to respond to failures in different time-frames. This gives us the opportunity to make trade-offs between accuracy, speed of detection and resource usage. For example in a cluster that runs a cloned application (i.e. a web server with replicated content), the reconfiguration triggered by a false suspicion is simple and can easily be undone when the system recognizes the mistake. However in the case of a partitioned database server a failure notification triggers a reconfiguration of the database layout and is likely to pull a standby server into the cluster configuration. Such a false suspicion is considered a disaster.

The multi-level failure detector has several modules, of which the most important are:

Gossip-based failure detector

The failure detector is part of the *light-weight state service* of SGCS, which provides functionality for participants to share individual state, the consistency of which is guaranteed through an epidemic protocol. A node includes in its state a version number which it increments each time is contacts another node to gossip to. In this gossip message the node will distribute a vector of node identifiers combined with the latest version numbers and a hash of the state information it has. The receiver of this message can decide, by comparing the version number and hash vectors, whether it has information that is newer than the sender, or that the sender has information that is newer. The receiver can then decide to push the newer information it has to the original gossiper, and to request transmission of the updated information from the gossiper. Failure detection can be performed without the need for an additional protocol: whenever a node receives a version number for a node that is larger than the one it has, it updates a local timestamp for that node. If for a certain amount of time the version number of a node has not been incremented, the node is declared to have failed. The rigorous mathematics unpinning the epidemic data dissemination theory allows for us to exactly determine the probability that a false suspicion will be made based on the parameters such as gossip rate and group size. As such this approach gives a clear control over the accuracy of the failure detection. A second advantage of this technique is that it allows each node to completely autonomously decide on whether another node has failed without the need for communication with other nodes.

Hierarchical gossip-based failure detector

Even though the gossip techniques are highly scalable, to minimize the communication and processing load needed to run the failure detector, a number of additional techniques are used. The SGCS light-weight state service has a notion of distance between nodes, which is partly based on operator configuration and partially determined through network tracking. The service will gossip less frequently with a node when the distance to that node increases. This has the additional advantage that router overload will not have an averse effect on the overall distribution probability. A second approach which yields an extremely scalable service is to organize the nodes into zones and organize the zones into virtual information hierarchies. The non-leaf zones in the hierarchy do not contain node information, but information on the child zones which are summaries from the individual entries at that zone level. This technique is successfully deployed in the Astrolabe tools and will be used for further integration into Galaxy [83].

Fast failure detector.

The gossip based failure detector is highly accurate, but slow. Over common communication links, with the communication load lower than 1% and a group size up to a thousand nodes, Galaxy tunes the failure detection threshold to be around 7-8 seconds. This is convenient as a number of anomalies in PC and operating systems architectures have shown that there are scenarios under which nodes can be network-silenced for several seconds, which would cause false suspicions when using more aggressive thresholds. Experiments with gossip-based failure detectors with more aggressive thresholds have been successfully performed, but mainly in controlled hardware and communication settings [79].

To address the need for faster failure detection, modules have been added to the multi-level failure detector that use more traditional techniques to track other nodes. These techniques are not as general and scalable as the gossip-based detector, and as such are only deployed on subsets of nodes. Most commonly used is the buddy failure-detector module where in a group the individual nodes will ping each other and the failure of a node to respond to pings will trigger a failure suspicion. This suspicion decision is broadcast to the other nodes. This module is fully configurable in how many nodes a single node will track and how many suspicions one needs to receive before the node will indicate the suspicion to the other system modules. The module makes use of the membership information produced by the light-weight state service to configure which nodes to track.

In Galaxy the farm membership is based on the gossip-based failure detection techniques, while the fast failure detectors are added to nodes based on management configuration. This can be based on knowledge of specific node and network configurations, or it can be part of a specific cluster profile.

Environmental failure detectors

The multi-level failure detector has a plug-in architecture for a class of very specialized failure detectors. These modules can track the environment to make intelligent and often highly accurate decisions about node availability. Examples of modules developed for particular Galaxy deployments are

- UPS management-event processors, which using the power supply information to determine the health of a node.
- SNMP monitors of switch link state information
- ICMP error message processors
- High-speed interconnect connection-state information

Consensus based membership

To support environments where agreement on the membership is essential, SGCS provides a consensus based membership protocol that runs a traditional leader-based consensus protocol. This membership module takes the suspicions generated by the fast or gossip-based failure detectors and runs an agreement protocol to ensure that all the participants in the group have seen the membership changes and have taken actions accordingly. If the SGCS stack instance is configured which virtual synchronous messaging, it will tag information onto the membership agreement messages to ensure agreement on message delivery with respect to the membership changes.

The use of different membership views in Galaxy

There are five different membership views in Galaxy, each of them implemented using some or all of the failure detectors described above.

1. *Farm Service membership*. This membership is based on the gossip based failure detector, and the cluster administrator configures the threshold of when to switch to the hierarchical gossip based failure detector. The cluster administrator also has the option to add fast and environmental failure detectors to the farm service but in general the gossip-based failure detector is sufficient to implement the farm service membership functionality.

2. *Cluster Service membership*. This service uses the membership information from the farm service as basic input and adds at least a fast failure detector to run within the cluster group, possibly augmented with the consensus based membership module. The exact modules to be used, including the additional environmental modules, are based on the cluster profile constructed by the administrator.

3. *Component membership*. The cluster service maintains the membership list for the local galaxy components that subscribe to the cluster service functionality, and exchanges this information with the other nodes in the cluster. Given that this shared state information is of the single-writer, multiple-reader kind, no special communication properties are required except for a reliable group multicast. When a new node joins the cluster it receives a state message from each of the other cluster nodes. The components receive a membership view which includes the state of each cluster node, information on which other nodes are running this particular component.

4. *Meta-cluster membership*. Each of the clusters in the meta-cluster group elect two nodes that will join a separate communication channel on which the elected nodes will exchange membership information from their local cluster. Associated with this communication channel is a failure detection module that runs less aggressive than the failure detectors in the member clusters. If at the local cluster the failure of an elected node is detected, a new node is elected to join the meta-cluster membership, before the meta-cluster failure detector is triggered. If the meta-cluster failure detector is triggered it is an indication that all the nodes in a member cluster have failed.

5. *Geoplex membership*. This membership protocol uses a technique similar to the meta-cluster. Each farm elects a number of nodes to participate in the geoplex

membership protocol, where upon node failure new nodes are elected to maintain the membership. The module is in general augmented with an environmental failure-detector that tracks the availability of the communication links between the data-centers.

8.5.2 Epidemic communication

Although a detailed description of the protocols used in SGCS is outside the scope of this chapter, it is important to understand the nature of the new protocols used to support the Galaxy operations. SGCS relies heavily on so-called epidemic or gossip protocols. In such protocols, each member, at regular intervals, chooses another member at random and exchanges information. Such information may include the "sequence number of the last message received", the "current view of the group", or the "version of the light-weight state". This gossip is known to spread exponentially fast to all members in spite of message loss and failed members, and therefore communication systems based on this scale extremely well. In fact, this dissemination process is a well-understood stochastic process about which we can make several probabilistic guarantees. It allows the protocol designer to provide guarantees such as "the probability that a message is not delivered atomically is less than epsilon". Here, epsilon can be configured to be a very small, configurable constant.

Next to the excellent scalability of the information dissemination techniques, protocols based on these techniques are also very robust to node failures and message loss. Once information has been exchanged with a few nodes, the probability that it will not reach all nodes eventually is very small. Experiments with 75% message loss, or with nodes perturbed 50% of the time, show that it is almost impossible to stop the information dissemination once a few nodes have been infected.

In SGCS the light-weight state service and the basic reliable multicast are implemented using epidemic protocols. These group membership protocols provide views to group members, which in turn, are built using information from the light-weight state service, and are thus also probabilistic in nature. On top of these protocols, SGSC provides lightweight versions of protocols that give more traditional (non-probabilistic) guarantees, such as virtual synchrony, total order and consensus. Because these protocols can now rely on the high probability of success, the amount of work they have to put into buffering, for example, is drastically reduced. These protocols do need to be aware that there are conditions under which the guarantees for success will be violated (i.e. a message will not be delivered or recovered by the reliable multicast protocol). These conditions however are detected by the lower-level protocols and indicated to its subscribers, which can then run recovery protocols if necessary.

8.6 The farm

Essential to enterprise cluster management is that the set of nodes out of which the clusters are created, can be viewed and managed as a single collection. This provides the manager with a single access point through which the operation of the complete data center clustering can be viewed and controlled. At this level Galaxy provides the notion of a Farm, which brings together all managed nodes into a single organizational unit.

To provide the farm abstraction, each node runs a farm service, which implements farm membership, node failure detection, intra-farm communication and state sharing. Additionally the farm service includes configuration and security services, as well as management functionality for control of a collection of farm service components. These components are separate from the farm service process, allowing for the set of services used in a farm to be configurable, and extendable. Farm service component must implement a service control interface used by the farm service process to control the service components and to establish a communication mechanism for components to communicate or exchange state.

The service components that are generally loaded at each farm node are:

- *Event Service*: this service monitors the state of local node and generates Farm Events, which are disseminated to the collection of nodes that make up the management cluster. At the management cluster the events are automatically processed, and saved for off-line auditing. The event processors in the management cluster have the ability to find correlations in generated events, and to generate operator warnings. Future work in this area will include a capability for the event processors to automate corrective actions for certain alarm conditions. We are looking for integration of our event technology with current advances in online data mining. The Event service is extensible and can use a variety of sources from which events can be drawn; current mechanisms include the Windows2000 Event log, WMI objects, performance counters and ODBC data sources. The set of information sources to monitor, as well which objects to collect from the sources is configurable. The service distributes its events in an XML encapsulation, allowing for easy extending at both event generator end process sides.

- *Measurement Service*: This service collects local performance information and posts this information in the Farm Management Information Database. The type and amount of information, as well as the update frequency is configurable at runtime when the information is drawn from the WMI and performance counters collections.
- *Process and Job Control Service*: This service implements functionality for controlled execution of processes on farm nodes. The service provides full security and job control, through an interface that allows authoritative users to create jobs on sets of nodes, add processes to an existing job and control the execution of the job. It provides full flexibility in client access. A client does not necessarily need to connect to a node where it wants to start processes, but can connect to any node in the farm to create and control jobs on any other node.
- *Remote Script Service*: Similar to the process service is the remote scripting engine, which allows administrators to produce simple scripts and execute these scripts concurrently on sets of nodes. This service is based on the Active Scripting component, which is augmented with a set of simple synchronization services, and output redirection and logging services.
- *Component Installation Service*: This is a simple service that ensures that the current components versions are installed based on farm configuration information and assigned cluster profiles.

8.6.1 Inserting nodes in the farm

To add a node to the farm, an administrator installs the basic farm management service and components, and starts the farm service. The farm service at the node uses a network resource discovery mechanism at startup time to locate basic information about the farm it is to be active in; it then announces itself to the farm controller service, which is part of the management cluster. The node will not become part of the farm automatically, but instead its insertion request is queued until an administrator has confirmed that this node is granted access to join the farm. Included in the confirmation process is a set of security actions that will enable the new node to access security tickets necessary for running a farm service, and to retrieve the necessary key information to communicate with the other farm members. After the new node has joined the farm it retrieves the farm configuration, which results in the launch of the service components.

8.6.2 The Farm Information Database

Information about the nodes in the farm is kept in the Farm Management Information Database. In part this database is filled with static, administrator supplied information about the nodes, but much of the information in the database is dynamic in nature and updated constantly by the farm management components such as the measurement and the process control services. The database is completely distributed in design and can be accessed from each of the nodes in the farm without the need to contact other nodes. The database is updated using an epidemic dissemination protocol to ensure scalable operation, automatically controlling the update rates and the amount of data transferred based on the number of nodes involved.

8.6.3 Landscapes

To manage very large farms Galaxy provides a mechanism for viewing the state of the farm using different *Landscapes*. Landscapes are employed as a mechanism for organizing nodes into smaller collections, such that the administrator can get a detailed view of parts of the farm without being forced to deal with the overall picture. Landscapes are created by specifying grouping criteria in the form of a SQL statement, which is applied to the Farm Management Information Database. There are two forms of landscapes, based on the actions that update the landscape grouping: If the data used is static or changes only infrequently, a trigger is added to landscape control components such that the landscape is recomputed whenever a field changes. The second form handles landscapes that provide views of data that is dynamic in nature, and where the landscape is recomputed on a periodic basis.

An example of the first form is grouping by physical location, by machine type or by assigned functionality. The second form generally handles grouping based on information such as compute load, number of active transaction components, number of client connections serviced or available storage space, often using an additional level of grouping based on cluster functionality.

8.7 Cluster design and construction

Although the farm provides the administrator with a set of new tools to manage collections of nodes, they are too primitive to support the level of control, service provision and customization needed in the types of cluster computing one typically encounters in an enterprise data center. To support the specialized operation each cluster instance requires additional functionality, which must be added on a per-application basis to nodes that make up a cluster1.

One of the most important tools in the farm management is the Cluster Designer, with which the administrator can group nodes together into a cluster and assign additional functionality in the form of cluster service components. Which components, and which component versions are needed for the operation of a particular cluster is specified in a predefined cluster profile, which is unique per type of cluster. The Cluster Designer manages these profiles, and a simple composition scheme is used to create new profiles for specialized clusters.

Basic in the operation of each cluster is a cluster service, which is a set of components running at each node in a cluster. The cluster service provides functionality similar to the farm service with which it cooperates in sharing network and state sharing resources. The cluster service augments the functionality of the farm service with additional failure detection, which is a more focused version that exploits knowledge about the cluster to provide an as fast and reliable service as is possible, and with a cluster membership abstraction. The cluster membership is maintained using a consensus based membership protocol, guaranteeing that every member sees identical changes in the membership in the same order. The cluster service also provides additional communication services to the cluster components, and it provides component control and membership functionality similar to that of the farm service. The cluster service makes the cluster, the cluster member nodes and its cluster service components available to its members through a single unified naming scheme.

Using the Cluster Designer is often not sufficient to create a fully functional cluster. Frequently the cluster service components need additional configuration. For example the web-clone cluster component needs to be configured to find the source of the document tree to be replicated or which cyclic multicast file transfer service to subscribe to, to receive document updates. Other more complex configuration situations arise when the nodes are physically sharing resources such as storage area networks, which may need additional management actions. Even though Galaxy is able to provide a large part of the cluster management and support infrastructure, the administrator still will need to install application level functionality and provide the configuration of application components.

1 In the metaphor of farms and landscapes, clusters can be visualized as pastures, and although we would have liked to use this term, we will continue to use cluster, as it contributes to a better understand of the work.

8.8 Cluster communication services

When a cluster-service instantiates the components it manages, it provides the components with an interface for intra-component communication and membership notification, similar to the service the farm-service provides to its components. The cluster-service communication facilities are different from those in the farm-service as they allow the cluster components to select communication properties such as reliability level and message ordering, or to provide flow and rate control parameterization. Cluster service components also have the possibility to create new communication channels within the cluster, to be used next to the basic intra-component channel.

The communication package used by the cluster-service is internally configurable, and the software components used inside the package are based on the particular cluster profile. For example if the cluster has access to multiple networks, or to a high-performance interconnect, it is the information in the cluster profile that determines which networks to use for administrative and intra-component communication, which networks can be used for additional communication created by the cluster service components, and which networks and techniques to use for failure detections.

8.9 Cluster profiles and examples

To support the initial Galaxy development goals we defined a number of cluster types and the cluster profiles that describe them. For two profiles we have developed a complete set of example cluster service components; the *Development Server* and *Component Server* are in daily use to support related research. The components of a third profile, the *Internet Game Server*, have been developed in the past months, and are currently under stress test. Of the other examples, the *Web-Clone* and the *Primary-Backup* profiles are making the transition from paper to real software design. An overview of the example cluster profiles is in table 8.1. In Figure 8.1 an example component layout is shown.

Profile	Cluster Service Components
Application Development Cluster	Process & job control, install & versioning, debugging and distributed logging, Visual Studio integration, resource measurement.
Component Management Cluster	System Services: Component Runtime, Component & Component factory membership, Debugging, Logging and Monitoring, Isolation Service
	Application Services: State management, Global snapshots & Check-pointing, Synchronization, Consensus, Voting, Shared Data Structures
Game Server	Cluster Client management, Application level request routing, Specialized Measurement service, Synchronization services, State Sharing services, Shared VM service
Primary-Backup Cluster	Distributed Log Service, VM replication Service, Transaction service, Lock Manager, Fail-over & reconfiguration service
MSCS – Compatible Cluster	Fail-over Manager, Resource Manager &Controllers, Node Manager, Membership Manager, Event Processor, Database Manager, Object Manager, Global Update Manager, Checkpoint Manager, Log Manager
Cloned Service Cluster	Fail-over & reconfiguration service, Version & install service, Multicast file transfer service, Cache Control management.
Partitioned Service Cluster	Client Request Redirector, Distributed Log service, Synchronization service, Transaction service, Lock Manager, Shared State Manager, Failover & reconfiguration service
Parallel Computing Cluster	Job Queuing Manager, Process & Job Control, Synchronization service, Logging & Debugging service.

Table 8.1. Example Cluster Profiles

8.9.1 Application development cluster

This is a cluster type that is in daily use within the research group, and handles the functionality that is needed to support team development of cluster applications. The majority of the functionality of the components deals with easy integration into a development environment, with support for process & job control, install & version control, logging and debug support, and resource usage measurement and reporting. Process control, install services and debug message services have client parts that are integrated into Microsoft Visual Studio, providing a single point of access for development of cluster applications. Resource usage information is accessed at the developer side through a version of the Windows NT task manager that provides a cluster view instead of single node view.

8.9.2 Component management cluster

One of the main objectives of our cluster area research is to develop a programming and execution environment for Cluster-Aware applications. This research is triggered by our past failures to provide useful high-level programming concepts for building advanced distributed systems. The services offered by the components in this cluster make up an execution runtime for cluster application implemented as components themselves, similar in concept to MTS or COM+/AppCenter [103].

The services are split into two categories: the first handles general management functionality: Component runtime management, which includes a functionality for providing component- & component factory membership, debugging, logging & monitoring services, and component isolation mechanisms for fault-containment. The second category handles the high-level distributed systems support for cluster-aware application construction: state management, global snapshots & check-pointing, synchronization, consensus, voting, shared data structures.

8.9.3 Game server cluster

One of the areas that provides us with the most complex technical challenges for providing scalable services is that of real-time multi-user Internet Game Servers. Clusters are an obvious match to provide a cost-effective “scale-out” solution, but the server side of the most games has not been designed with any cluster-awareness. In our research we are interested in investigating what support services are useful to provide on a cluster dedicated to serving Game Engines. Our first experiments involved restructuring one of the Quake server engines and build service components that allowed load-balancing as well as client fail-over. The initial results are promising, but the testing (simulating thousands of Quake clients) is still too difficult to report on solid results.

The cluster services that are developed to support the cluster version of the Quake game server are: client management, application level request routing, dedicated load measurement service, synchronization and state sharing services, and a shared VM service.

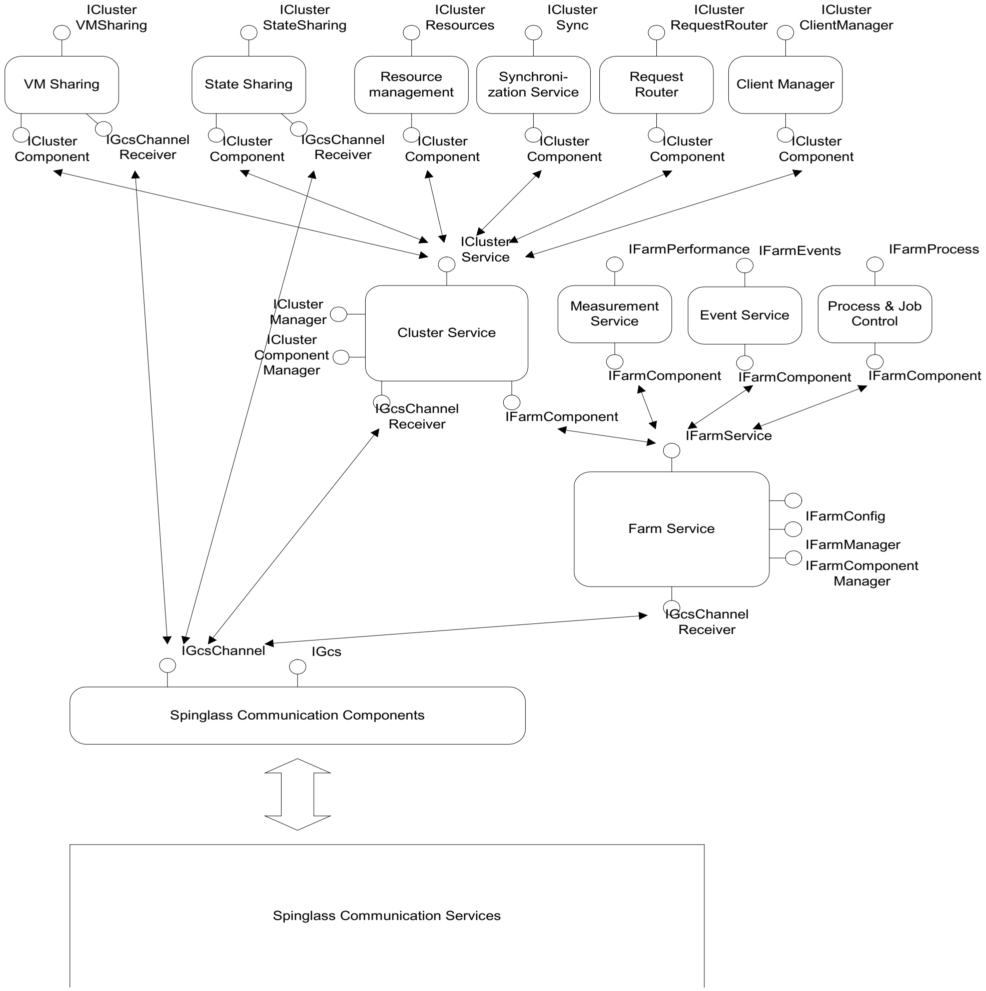


Figure 8.1. Layout of the various components in the Game Server Profile

8.10 Component development libraries

Building farm and cluster components and their clients is made easier through the use of two libraries that we have developed. The first library supports the creation of the cluster components by providing classes that handle the interaction with the farm and cluster server, by parsing membership reports and incoming message types and providing those through event based processing structure. This library also supplies classes for handling clients from outside the cluster, in case this component supports direct connections made by client programs. It provides a scheduler centralizing the handling of input from the client, the cluster-services, and other component instances communicating through the cluster service, making it easier to building state machine structured distributed components.

The second library provides a collection of classes that makes it easier to build client applications that do not run on the clusters themselves. The classes encapsulate connection management such that a cluster name can be used at connection time, and the support classes manage the connection to the nodes in the named cluster, and automatically reconnecting on connection failure, possibly to another node in the cluster. There are also classes in both libraries that deal with forwarding membership information to clients, naming clients, and routing replies to client programs.

8.11 Related work

There is a significant body of work on cluster management systems, but none of the previous work provides a framework for farm management and supports for multiple styles as cluster computing in the way that Galaxy does. Most of the cluster management systems provide only support for very specific styles of management as they are built to support the operation of specific commercial clusters. Of these commercial clusters *VAXClusters* [58] and *Parallel Sysplex* [70] deserve special mention as they did groundbreaking work in building general distribution support for clusters. *Microsoft's Cluster Service* (MSCS) [100] has brought enterprise class clustering to the masses through a low-cost solution for mainly fail-over scenarios. MSCS is limited in scale [101] and other styles of clustering are offered through separate Microsoft products such as *NLB* for web-site clustering or *AppServer* for middle-tier cluster management. In the Open Source community large number of packages are under development that support enterprise computing, of which two receive most attention: *Linux-HA* [85] which provides failure detection using a heart beat mechanism for Linux machines, and Compac's Cluster-Infrastructure for Linux [20] which provides a re-implementation of some of the Tandem NonStop Cluster membership and inter-node communication technology for Linux machines.

Cluster management systems supporting compute clusters are ubiquitous, most of them built at the major research labs to support custom clusters constructed over the past decade, e.g. [39]. With the advent of a trend towards more cost effective parallel computing, the *Beowulf* [97] package for parallel computing under Linux has become very popular, supporting a large number of compute cluster over the world. New directions in parallel computing require a more fine-grained, component based approach, as found in NCSA's *Symera* [40].

Historically there were only a few high-profile management systems for high-available cluster computing, which often was supported by dedicated cluster hardware support such as the systems by Tandem and Stratus [49]. Since the move to more cost enterprise systems most of the vendors have developed commercial cluster management products for high availability. All of these systems have limited scalability [28] and are very specific in terms of hardware support or supported configurations. An overview of the various vendor products can be found in [77].

Scalability in cluster systems has been addressed in cluster research, but mostly from an application specific viewpoint: The *Inktomi* [42] technology is a highly scalable cluster targeted towards Internet search engines and document retrieval. A cluster based SS7 call processing center [43] with good scalability from a real-time perspective, was build in 1998 using Cornell communication technology. Another cluster system that claims excellent scalability is *Porcupine*, which is very specifically targeted towards Internet mail processing systems [88]. In terms of specific distributed systems support for cluster management systems, there is only limited published work. Next to the systems based on Cornell research technology, the *Phoenix* system from IBM research has similar properties and is also applied to cluster management [3]. In the *Oceano* project research at IBM have started to experiment with automatic configuration and management of large datacenters, the results of which are fueling the *Autonomic* computing projects that focus on automatic management and self-healing systems.

8.12 Evaluating scalability

One of our activities that has not received much attention in this chapter but is a very important part of our research is ongoing work on developing a systematic approach to evaluating scalability. In this chapter we have made a number of claims about the scalability of the Galaxy farm and cluster components based on our theoretical and experimental results. It is outside of the scope of this chapter to present the detailed experimental and usage results and we refer the interested reader to the papers on probabilistic multicast [10,11], buffering and garbage collection techniques [72], gossip based failure detection [81] and hierarchical epidemics & scalable datafusion [82,83] for additional details.

There are many aspects to the scalability of distribution services. More research is needed before we are able to identify, isolate and measure, in a scientific manner, the scalability of systems. Providing a framework for reasoning about the scalability of complex distributed systems such as Galaxy, is one of our highest research priorities.

PART - III

System experimentation and analysis of large-scale systems

Introduction

Understanding the way systems are used in production settings is essential for successful systems research. The design and development of Galaxy and the other tools that came out of the *Spinglass* project was largely driven by the experiences with deploying complex software systems that were a part of earlier research prototypes or widely used commercial products.

Observing active production systems in a way that lets us draw valid conclusions from the observations is a major challenge. Unlike a more traditional experiment design, there is only limited control over the variables that determine the usage of the system. Instrumentation, data collection and data analysis become very complex operations, and frequently a number of iterations of all three phases is necessary before the observation system meets the rigid requirements of proper system analysis.

During 1997 and 1998, I designed a system for large scale tracing of file systems usage on standard workstations. The work was triggered by the observation that it was almost 10 years since the last general file system trace experiments were published, and that the changes in both workstation hardware and computer usage no longer warranted the usage of those older traces as predictors for modern system usage. Thus we are building cluster out of components which have a poorly understood behavior. A second motivation for the work was to understand the challenges in the design of a scalable trace system that would collect the data in a rigorous scientific manner. Analysis of the older traces has shown that those experiments were flawed by the lack of appropriate statistical analysis, and the absence of the relevant information from the traces.

It took more than a year to refine the trace system and the analysis procedures to a point where large scale tracing could be executed with confidence and the results used for proper statistical analysis. The data collection phase resulted in almost 20 GByte of data representing over 190 million trace records. After the collection phase it still took more than 8 months, and the use of advanced tools, to produce the first results that could be seen as statistically significant. The sheer amount of trace data required a non-traditional approach in processing the data. The results of the study were published in the 1999 ACM Symposium on Operating Systems Principles (SOSP) and can be found in chapter 9.

The case for rigorous statistical analysis

Section 6 of chapter 9 goes into details about the correctness of the statistical analysis and emphasizes the importance of including tail-analysis in the experiment analysis. Frequently, experimental results are described using simple statistical tools (average, mean, variation, 95% mark) without actually testing whether the data distribution as a whole meets the criteria of a normal distribution. The presence of heavy tails in a workload can have a significant influence on the operation of a system, but by only using simple statistical tests it is likely that a tail is never discovered. This will lead to erroneous conclusions or under-performing systems if only the simple statistical parameters are used to tune a system.

In the case of the file systems tracing, the statistical analysis process is complicated by the difficulties of analyzing large amounts of trace data. In the statistical process it is essential that the distributions are visually inspected for anomalies, which becomes impossible when there are so many possible relations and time-scales to explore. Although there aren't many precendents in systems research of processing these large amounts of data for exploring potential relations, it is a process that is quite common in settings outside of research. The process of exploring the trace-data is similar to the data-mining performed by many corporations on their customer and sales databases. By switching to using commercial-grade Online Analytical Processing (OLAP) tools and storing the trace data in a high-performance commercial database, the processing could be done with confidence. So-called "drill-down" operations into the many data-cubes constructed out of the data, provided clear insights and permitted us to reach conclusions that would not have been possible without the use of these tools.

A final conclusion

The research cited above was primarily focused on the construction of the experiments themselves and the analysis of the results., Thought should also be given to how the results of these experiments might drive a new research agenda for file system development.

If there is one clear observation on the differences between the earlier traces and the new results it is that applications on the early Unix systems were quite uniform in the way that they used the file system. In general they were developed by a small group of developers who shared the same philosophy of application design. The newer traces show that there is hardly any uniformity in the way that modern applications use the file system. The developers are a large, diverse group with no common approach. One application will close files immediately after reading them into memory, while others may keep them open from complete duration of the applications lifetime. This leads to the conclusion that it is almost impossible to construct generic workloads that one can use to test file-system effectiveness. Using application centered workloads is likely to be more successful and produce more useful insights.

An important research question to ask about the observation of lack of uniformity in file systems programming is what triggers this diverse behavior. Could it be that the limited functionality in the common file access application programming interface (open-rw-close) also is a contributing factor to the chaos? An avenue to explore for the future is whether a radical departure from the way developers use file access may better serve the needs of applications developers and will provide file systems with more predictable application behavior.

Chapter 9

File system usage in Windows NT 4.0

We have performed a study of the usage of the Windows NT File System through long-term kernel tracing. Our goal was to provide a new data point with respect to the 1985 and 1991 trace-based File System studies, to investigate the usage details of the Windows NT file system architecture, and to study the overall statistical behavior of the usage data.

In this chapter we report on these issues through a detailed comparison with the older traces, through details on the operational characteristics and through a usage analysis of the file system and cache manager. Next to architectural insights we provide evidence for the pervasive presence of heavy-tail distribution characteristics in all aspect of file system usage. Extreme variances are found in session inter-arrival time, session holding times, read/write frequencies, read/write buffer sizes, etc., which is of importance to system engineering, tuning and benchmarking.

9.1 Introduction

There is an extensive body of literature on usage patterns for file systems [5,45,59,66,71], and it has helped shape file system designs [56,69,87] that perform quite well. However, the world of computing has undergone major changes since the last usage study was performed in 1991; not only have computing and network capabilities increased beyond expectations, but the integration of computing in all aspects of professional life has produced new generations of systems and applications that no longer resemble the computer operations of the late eighties. These changes in the way computers are used may very well have an important impact on the usage of computer file systems.

One of the changes in systems has been the introduction of a new commercial operating system, Microsoft's Windows NT, which has acquired an important portion of the professional OS market. Windows NT is different enough from Unix that Unix file systems studies are probably not appropriate for use in designing or optimizing Windows NT file systems.

These two observations have lead us to believe that new data about file systems usage is required, and that it would be particularly interesting to perform the investigation on a Windows NT platform.

In this chapter we report on a file system usage study performed mainly during 1998 on the Windows NT 4.0 operating system. We had four goals for this study:

1. Provide a new data point with respect to earlier file system usage studies, performed on the BSD and Sprite operating systems.
2. Study in detail the usage of the various components of the Windows NT I/O subsystem, and examine undocumented usage such as the *FastIO* path.
3. Investigate the complexity of Windows NT file system interactions, with a focus on those operations that are not directly related to the data path.
4. Study the overall distribution of the usage data. Previous studies already hinted at problems with modeling outliers in the distribution, but we believe that this problem is more structural and warrants a more detailed analysis.

Next to these immediate goals, we wanted the investigation to result in a data collection that would be available for public inspection, and that could be used as input for file system simulation studies and as configuration information for realistic file system benchmarks.

The complexity of Windows NT file usage is easily demonstrated. When we type a few characters in the Notepad text editor, saving this to a file will trigger 26 system calls, including 3 failed open attempts, 1 file overwrite and 4 additional file open and close sequences.

The rest of this chapter is structured as follows: in section 9.2 we describe the systems we measured, and in section 9.3 and 9.4, we describe the way we collected the data and processed it. In section 9.5 we examine the file system layout information, and in section 9.6 we compare our tracing results with the BSD and Sprite traces. Section 9.7 contains a detailed analysis of the distribution aspects of our collected data.

In comparison with the Sprite and BSD traces
– Per user throughput remains low, but is about 3 times higher (24 Kbytes/sec) than in Sprite (8 Kb/sec) – Files opened for data access are open for increasingly shorter periods: 75% of files remain open for less then 10 milliseconds versus a 75th percentile of 250 milliseconds in Sprite. – Most accessed files are short in length (80% are smaller than 26 Kbytes), which is similar to Sprite. – Most access (60%) to files is sequential, but there is a clear shift towards random access when compared to Sprite. – The size of large files has increased by an order of magnitude (20% are 4Mbytes or larger), and access to these files accounts for the majority of the transferred bytes. – 81% of new files are overwritten within 4 milliseconds (26%) or deleted within 5 seconds (55%).
Trace data distribution characteristics
– There is strong evidence of extreme variance in all of the traced usage characteristics. – All the distributions show a significant presence of heavy-tails, with values for the Hill estimator between 1.2 and 1.7, which is evidence of infinite variance. – Using Poisson processes and Normal distributions to model file system usage will lead to incorrect results.
Operational characteristics
– The burstiness of the file operations has increased to the point where it disturbs the proper analysis of the data. – Control operations dominate the file system requests: 74% of the file opens are to perform a control or directory operation. – In 60% of the file read requests the data comes from the file cache. – In 92% of the open-for-read cases a single prefetch was sufficient to load the data to satisfy all subsequent reads from the cache. – The FastIO path is used in 59% of the read and 96% of the write requests. – Windows NT access attributes such as *temporary file*, *cache write- through, sequential access only*, can improve access performance significantly but are underutilized.
File system content
– Executables, dynamic loadable libraries and fonts dominate the file size distribution. – 94% of file system content changes are in the tree of user *profiles* (personalized file cache). – Up to 90% of changes in the user's profile occur in the WWW cache. – The time attributes recorded with files are unreliable

Table 9.1. Summary of observations

Sections 9.8, 9.9 and 9.10 contain details about the operation of various Windows NT file system components. Section 9.11 touches on related work and section 9.12 summarizes the major points of the study. An overview of our observations can be found in table 9.1.

9.2 Systems under study

We studied a production environment in which five distinct categories of usage are found:

- **Walk-up usage**. Users make use of a pool of available systems located in a central facility. The activities of these users vary from scientific analysis and program development to document preparation.
- **Pool usage**. Groups of users share a set of dedicated systems, located near their work places. These users mainly are active in program development, but also perform a fair share of multimedia, simulation and data processing.
- **Personal usage**. A system is dedicated to a single user and located in her office. The majority of the activities is in the category of collaborative style applications, such as email and document preparation. A smaller set of users uses the systems for program development.
- **Administrative usage**. All these systems are used for a small set of general support tasks: database interaction, collaborative applications, and some dedicated administrative tools.
- **Scientific usage**. These systems support major computational tasks, such as simulation, graphics processing, and statistical processing. The systems are dedicated to the small set of special applications.

The first four categories are all supported by Pentium Pro or Pentium II systems with 64-128 Mb memory and a 2-6 GB local IDE disk. The *pool* usage machines are in general more powerful (300-450 MHz, some dual processors), while the other machines are all in the 200 MHz range. The *scientific* usage category consists of Pentium II 450 Xeon dual and quad processors with a minimum of 256 MB of memory and local 9-18 GB SCSI Ultra-2 disks. All systems ran Windows NT 4.0 with the latest service packs applied. At the time of the traces the age of file systems ranged from 2 months to 3 years, with an average of 1.2 years.

There is central network file server support for all users. Only a limited set of personal workstations is supported through a backup mechanism, so central file storage is implicitly encouraged. All systems are connected to the network file servers through a 100 Mbit/sec switched Ethernet. The users are organized in three different NT domains, one for the walk-up usage, one general usage and one for experiments. The experimental domain has a trust relationship with the general domain and network file services are shared. The walk-up domain is separated from the other domains through a network firewall and has its own network file services.

From the 250 systems that were available for instrumentation, we selected a set of 45 systems based on privacy concerns and administrative accessibility. A subset of these systems was traced for 3 periods of 2 weeks during the first half of 1998 while we adjusted the exact type and amount of data collected. Some of the changes were related to the fact that our study was of an exploratory nature and the data collection had to be adjusted based on the initial results of the analysis. Other adjustments were related to our quest to keep the amount of data per trace record to an absolute minimum, while still logging sufficient information to support the analysis. We were not always successful as, for example, logging only the read request size is of limited use if the bytes actually read are not also logged. The analysis reported in this chapter is based on a final data collection that ran for 4 weeks in November and December of 1998. The 45 systems generated close to 19 GB of trace data over this period.

Since then we have run additional traces on selected systems to understand particular issues that were unclear in the original traces, such as burst behavior of paging I/O, reads from compressed large files and the throughput of directory operations.

9.3 Collecting the data

The systems were instrumented to report two types of data: 1) snapshots of the state of the local file systems and 2) all I/O requests sent to the local and remote file systems. The first type is used to provide basic information about the initial state of the file system at the start of each tracing period and to establish the base set of files toward which the later requests are directed. In the second type of data all file system actions are recorded in real-time.

On each system a trace agent is installed that provides an access point for remote control of the tracing process. The trace agent is responsible for taking the periodic snapshots and for directing the stream of trace events towards the collection servers. The collection servers are three dedicated file servers that take the incoming event streams and store them in compressed formats for later retrieval. The trace agent is automatically started at boot time and tries to connect to a collection server; if it succeeds, it will initiate the local data collection. If a trace agent loses contact with the collection servers it will suspend the local operation until the connection is re-established.

9.3.1 File system snapshots

Each morning at 4 o'clock a thread is started by the trace agent server to take a snapshot of the local file systems. It builds this snapshot by recursively traversing the file system trees, producing a sequence of records containing the attributes of each file and directory in such a way that the original tree can be recovered from the sequence. The attributes stored in a *walk* record are the file name and size, and the creation, last modify and last access times. For directories the name, number of files entries and number of subdirectories is stored. Names are stored in a short form as we are mainly interested in the file type, not in the individual names. On FAT file systems the creation and last access times are not maintained and thus ignored.

The trace agent transfers these records to the trace collection server, where they are stored in a compressed format. Access to the collection files is through an OLE/DB provider, which presents each file as two database tables: one containing the directory and the other containing file information.

Producing a snapshot of a 2 GB disk takes between 30 and 90 seconds on a 200 MHz P6.

9.3.2 File system trace instrumentation

To trace file system activity, the operating system was instrumented so that it would record all file access operations. An important subset of the Windows NT file system operations are triggered by the virtual memory manager, which handles executable image loading and file cache misses through its memory mapped file interface. As such, it is not sufficient to trace at the system call level as was done in earlier traces. Our trace mechanism exploits the Windows NT support for transparent layering of device drivers, by introducing a filter driver that records all requests sent to the drivers that implement file systems. The trace driver is attached to each driver instance of a local file system (excluding removable devices), and to the driver that implements the network redirector, which provides access to remote file systems through the CIFS protocol.

All file systems requests are sent to the I/O manager component of the Windows NT operating system, regardless of whether the request originates in a user-level process or in another kernel component, such as the virtual memory manager or the network file server. After validating the request, the I/O manager presents it to the top-most device-driver in the driver chain that handles the volume on which the file resides. There are two driver access mechanisms: one is a generic packet based request mechanism, in which the I/O manager sends a packet (an *IRP* -- I/O request packet) describing the request, to the drivers in the chain sequentially. After handling a request packet a driver returns it to the I/O manager, which will then send it to the next device. A driver interested in post-processing of the request, after the packet has been handled by its destination driver, modifies the packet to include the address of a callback routine. A second driver access mechanism, dubbed *FastIO*, presents a direct method invocation mechanism: the I/O manager invokes a method in the topmost driver, which in turn invokes the same method on the next driver, and so on. The FastIO path is examined in more detail in section 9.10.

The trace driver records 54 IRP and FastIO events, which represent all major I/O request operations. The specifics of each operation are stored in fixed size records in a memory buffer, which is periodically flushed to the collection server. The information recorded depends on the particular operation, but each record contains at least a reference to the file object, IRP, File and Header Flags, the requesting process, the current byte offset and file size, and the result status of the operation. Each record receives two timestamps: one at the start of the operation and the other at completion time. These time stamps have a 100 nanosecond granularity. Additional information recorded depends on the particular operation, such as offset, length and returned bytes for the read and write operations, or the options and attributes for the create operation. An additional trace record is written for each new file object, mapping object id to a file name.

The trace driver uses a triple-buffering scheme for the record storage, with each storage buffer able to hold up to 3,000 records. An idle system fills this size storage buffer in an hour; under heavy load, buffers fill in as little as 3-5 seconds. Were the buffers to fill in less than 1 second, the increased communication latency between the host and server could lead to the overflow of the tracing module storage buffer. The trace agent would detect such an error, but this never occurred during our tracing runs.

Kernel profiling has shown the impact of the tracing module to be acceptable; under heavy IRP load the tracing activity contributed up to 0.5% of the total load on a 200 MHz P6.

In a 24-hour period the file system trace module would record between 80 thousand and 1.4 million events.

9.3.3 Executable and paging I/O

Windows NT provides functionality for memory mapped files, which are used heavily by two system services: (1) the loading of executables and dynamic loadable libraries is based on memory mapped files, and (2) the cache manager establishes a file mapping for each file in its cache, and uses the page fault mechanism to trigger the VM manager into loading the actual data into the cache. This tight integration of file system, cache manager and virtual memory manager poses a number of problems if we want to accurately account for all the file system operations.

The VM manager uses IRPs to request the loading of data from a file into memory and these IRPs follow the same path as regular requests do. File systems can recognize requests from the VM through a *PagingIO* bit set in the packet header. When tracing file systems one can ignore a large portion of the paging requests, as they represent duplicate actions: a request arrives from process and triggers a page fault in the file cache, which triggers a paging request from the VM manager. However, if we do ignore paging requests we would miss all paging that is related to executable and dynamic loadable library (dll) loading, and other use of memory mapped files. We decided to record all paging requests and filter out the cache manager induced duplicates during the analysis process.

We decided in favor of this added complexity, even though it almost doubled the size of our traces, because of the need for accuracy in accounting for executable-based file system requests. In earlier traces the *exec* system call was traced to record the executable size, which was used to adjust the overall trace measurements. In Windows NT this is not appropriate because of the optimization behavior of the Virtual Memory manager: executable code pages frequently remain in memory after their application has finished executing to provide fast startup in case the application is executed again.

9.3.4 Missing and noise data

We believe the system is complete in recording all major file system IO events, which is sufficient for getting insight into general Windows NT file system usage. There are many minor operations for which we did not log detailed information (such as locking and security operations), but they were outside of the scope of our study. During our analysis we found one particular source of noise: the local file systems can be accessed over the network by other systems. We found this access to be minimal, as in general it was used to copy a few files or to share a test executable. Given the limited impact of these server operations we decided to not remove them from the trace sets.

9.4 The data analysis process

The data analysis presented us with a significant problem: the amount of data was overwhelming. The trace data collection run we are reporting on totaled close to 20 GB of data, representing over 190 million trace records. The static snapshots of the local disks resulted in 24 million records.

Most related tracing research focuses on finding answers to specific sets of questions and hypotheses, which could be satisfied through the use of extensive statistical techniques, reducing the analysis to a number crunching exercise. Given the exploratory nature of our study we needed mechanisms with which we could *browse* through the data and search for particular patterns, managing the exposed level of details. Developing a representation of the data such that these operations could be performed efficiently on many millions of records turned out to be a very hard problem.

We were able to find a solution by realizing that this was a problem identical to the problems for which there is support in data-warehousing and on-line analytical processing (OLAP). We developed a de-normalized star schema for the trace data and constructed corresponding database tables in SQL-server 7.0. We performed a series of summarization runs over the trace data to collect the information for the dimension tables. Dimension tables are used in the analysis process as the category axes for multi-dimensional cube representations of the trace information. Most dimensions support multiple levels of summarization, to allow a *drill-down* into the summarized data to explore various levels of detail. An example of categorization is that a mailbox file with a *.mbx* type is part of the *mail files* category, which is part of the *application files* category.

We departed from the classical data-warehouse model in that we used two fact tables (the tables that hold all the actual information), instead of one. The first table (*trace*) holds all the trace data records, with key references to dimension tables. The second table (*instance*) holds the information related to each FileObject instance, which is associated with a single file open-close sequence, combined with summary data for all operations on the object during its life-time. Although the second table could be produced by the OLAP system, our decision to use two fact tables reduced the amount of storage needed in the trace table by references to the instance table, reducing the processing overhead on operations that touch all trace records.

The use of a production quality database system provided us with a very efficient data storage facility. An operation that would touch all trace data records, such as calculation of the basic statistical descriptors (avg, stdev, min, max) of request inter-arrival times, runs at 30% of the time a hand optimized C-process on the original trace data takes. More complex statistical processing that could not be expressed in SQL or MDX was performed using the SPSS statistical processing package that directly interfaces with the database.

Because we processed the data using different category groupings (e.g. per day, per node, per user, per process, etc.) our analysis frequently did not result in single values for the statistical descriptors. In the text we show the ranges of these values or, where relevant, only the upper or lower bound.

9.5 File system content characteristics

To accurately analyze the real-time tracing results we needed to examine the characteristics of the set of files that were to be accessed in our trace sessions. For this we took snapshots of each of the file systems that was used for tracing as described in section 9.3.1. We collected file names and sizes, and creation and access times, as well as directory structure and sizes.

We supplemented this data with periodic snapshots of the user directories at the network file severs. However, the results of these snapshots cannot be seen as the correct state of the system from which the real-time tracing was performed, as they included the home directories of more users than those being traced. The network file server information was used to establish an intuitive notion of the differences between local and network file systems.

Recently Douceur and Bolosky have published a study on the content of over 10,000 Windows NT file systems within Microsoft [28]. Most of our findings are consistent with their conclusions, and we refer to their paper for a basic understanding of Windows NT file system content. Our content tracing allowed us to track the state of the file systems over time, and in this section we report from that specific view.

We see that the local file systems have between 24,000 and 45,000 files, that the file size distribution is similar for all systems, and that the directory depth and sizes are almost identical. File systems are between 54% and 87% full.

The network server file systems are organized into *shares*, which is a remote mountable sub-tree of a file system. In our setting each share represents a user's home directory. There was no uniformity in size or content of the user shares; sizes ranged from 500 Kbytes to 700 Mbytes and number of files from 150 to 27,000. The directory characteristics exhibit similar variances.

Decomposition of the local and network file systems by file type shows a high variance within the categories as well as between categories. What is remarkable is that this file type diversity does not appear to have any impact on the file size distribution; the large-sized outliers in the size distribution dominate the distribution characteristics. If we look again at the different file types and weigh each type by file size we see that the file type distribution is similar for all system types, even for the network file systems. The file size distribution is dominated by a select group of file-types that is present in all file systems. For local file systems the size distribution is dominated by executables, dynamic loadable libraries and fonts, while for the network file system the set of large files is augmented with development databases, archives and installation packages.

When we examine the local file systems in more detail we see that the differences in file system state are determined by two factors: 1) the file distribution within the user's *profile*, and 2) the application packages installed. Most of our test systems have a limited number of user specific files in the local file system, which are generally stored on the network file servers.

Of the user files that are stored locally between 87% and 99% can be found in the *profile* tree (\winnt\profiles\<username>). Each profile holds all the files that are unique to a user and which are stored by the system at a central location. These files are downloaded to each system the user logs into from a profile server, through the *winlogon* process. This includes files on the user's desktop, application specific data such as mail files, and the user's world-wide-web cache. At the end of each session the changes to the profiles are migrated back to the central server. When we detect major differences between the systems, they are concentrated in the tree under the \winnt\profiles directory. For the "*Temporary Internet Files*" WWW cache we found sizes between 5 and 45 Mbytes and with between 2,000 and 9,500 files in the cache.

A second influence on the content characteristics of the file system is the set of application packages that are installed. Most general packages such as Microsoft Office or Adobe Photoshop have distribution dynamics that are identical to the Windows NT base system and thus have little impact on the overall distribution characteristics. Developer packages such as the Microsoft Platform SDK, which contains 14,000 files in 1300 directories, create a significant shift in file-type count and the average directory statistics.

When we examine the changes in the file systems over time, similar observations can be made. Major changes to a Windows NT file system appear when a new user logs onto a system, which triggers a profile download, or when a new application package is installed. Without such events, almost all of the measured changes were related to the current user's activities, as recorded in her profile. A commonly observed daily pattern is one where 300-500 files change or are added to the system, with peaks of up to 2,500 and 3,000 files, up to 93% of which are in the WWW cache.

Changes to user shares at the network file server occur at much slower pace. A common daily pattern is where 5-40 files change or are added to the share, with peaks occurring when the user installs an application package or retrieves a large set of files from an archive.

If we look at the age of files and access patterns over time, a first observation to make is that the three file times recorded with files (creation, last access, last change) are unreliable. These times are under application control, allowing for changes that cause inconsistencies. For example, in 2-4% of the examined cases, the last change access is more recent than the last access times. Installation programs frequently change the file creation time of newly installed files to the creation time of the file on the installation medium, resulting in files that have creation times of years ago on file systems that are only days or weeks old. In [89] the creation times were also not available and a measure used to examine usage over time was the *functional lifetime*, defined as the difference between the last change and the last access. We believe that these timestamps in Windows NT are more accurate than the creation time, as the file system is the main modifier of these timestamps, but we are still uncertain about their correctness, and as such we cannot report on them.

9.6 BSD & Sprite studies revisited

One of the goals of this study was to provide a new data point in relation to earlier studies of the file system usage in BSD 4.2 and the Sprite operating systems. These studies reported their results in three categories: 1) user activity (general usage of the file system on a per user basis), 2) access patterns (read/write, sequential/random), and 3) file lifetimes. A summary of the conclusions of this comparison can be found in table 9.1.

The Windows NT traces contain more detail than the BSD/Sprite traces, but in this section we will limit our reporting to the type of data available in the original studies.

Strong caution: when summarizing the trace data to produce tables identical to those of the older traces, we resort to techniques that are **not** statistically sound. As we will show in section 9.7, access rates, bytes transferred and most of the other properties investigated are not normally distributed and thus cannot be accurately described by a simple average of the data. We present the summary data in table 9.2 and 9.3 to provide a historical comparison.

9.6.1 User activity

Table 9.2 reports on the user activity during the data collection. The tracing period is divided into 10-minute and 10-second intervals, and the number of active users and the throughput per user is averaged across those intervals. In the BSD and Sprite traces it was assumed that 10 minutes was a sufficient period to represent a steady state, while the 10-second average would more accurately capture bursts of activity.

The earlier traces all reported on multi-user systems, while the Windows NT systems under study are all configured for a single user. A user and thus a system are considered to be active during an interval if there was any file system activity during that interval that could be attributed to the user. In Windows NT there is a certain amount of background file system activity, induced by systems services, that was used as the threshold for the user activity test.

Interval	*User Activity*	*Windows NT*	*Sprite*	*BSD*
10-minute intervals	Max Number of active users	45	27	31
	Avregae number of active users	28.9 (21.6)	9.1 (5.1)	12.6
	Average Throughput for a user in an interval	24.4 (57.9)	8.0 (36)	0.4 (0.4)
	Peak throughput for an active user	814	458	NA
	Peak Throughput system wide	814	681	NA
10-second intervals	Max Number of active users	45	12	NA
	Avregae number of active users	6.3 (15.3)	1.6 (1.5)	2.5 (1.5)
	Average Throughput for a user in an interval	42.5 (191)	47.0 (268)	1.5 (808)
	Peak throughput for an active user	8910	9871	NA
	Peak Throughput system wide	8910	9977	NA

Table 9.2. User activity. The throughput is reported in Kbytes/second (with the standard deviation in parentheses)

File Usage	*Accesses (%)*				*Bytes(%)*				*Type of transfer*	*Accesses (%)*				*Bytes (%)*			
	W	-	+	*S*	*W*	-	+	*S*		*W*	-	+	*S*	*W*	-	+	*S*
Read-on;y	79	21	97	88	59	21	99	80	Whole file	68	1	99	78	58	3	96	89
									Other Sequential	20	0	62	19	11	0	72	5
									Random	12	0	99	3	31	0	97	7
Write-only	18	3	77	11	26	0	73	19	Whole file	78	5	99	67	70	1	99	69
									Other Sequential	7	0	51	29	3	0	47	19
									Random	15	0	94	4	27	0	99	11
Read/Write	3	0	16	1	15	0	70	1	Whole file	22	0	90	0	5	0	76	0
									Other Sequential	3	0	28	0	0	0	14	0
									Random	74	2	100	100	94	9	100	0

Table 9.3. Access patterns, the *W* column holds the mean for the Windows NT tracses, The *S* columns holds the values from the Sprite traces. The - and + columns indicate the range for the WIndows NT values. All numbers are reported in percentages

The result of the comparison is in table 9.2. The average throughput per user has increased threefold since the 1991 Sprite measurements. A remarkable observation is that this increase can only be seen for the 10-minute periods, for the 10-second period there was no such increase and the peak measurements are even lower.

The Sprite researchers already noticed that the increase in throughput per user was not on the same scale as the increase of processor power per user. They attributed this to the move from a system with local disk to a diskless system with network file systems. In our traces we are able to differentiate between local and network access, and when summarizing it appears that there is indeed such a difference in throughput. However detailed analysis shows that the difference can be completely attributed to the location of executables and large system-files such as fonts, which are all placed on the local disk.

One of the reasons for the high peak load in Sprite was the presence of large files from a scientific simulation. Although the scientific usage category in our traces uses files that are of an order of magnitude larger (100-300 Mbytes), they do not produce the same high peak loads seen in Sprite. These applications read small portions of the files at a time, and in many cases do so through the use of memory-mapped files.

The peak load reported for Windows NT was for a development station, where in a short period a series of medium size files (5-8 Mb), containing precompiled header files, incremental linkage state and development support data, was read and written.

9.6.2 File access patterns

The BSD and Sprite (and also the VMS [78]) traces all concluded that most access to files is sequential. Summaries of our measurements, as found in table 9.3, support this conclusion for Windows NT file access, but there is also evidence of a shift towards more randomized access to files when compared to the Sprite results.

A sequential access is divided into two classes: complete file access and partial file access. In the latter case all read and write accesses are sequential but the access does not start at the beginning of the file or transfers fewer bytes than the size of the file at close time.

The Windows NT traces do not support the trend seen in Sprite, where there was a 10% increase in sequential access. On average 68% of the read-only accesses were whole-file sequential, versus 78% in the Sprite traces. A significant difference from Sprite is the amount of data transferred sequentially: in Sprite 89% of the read-only

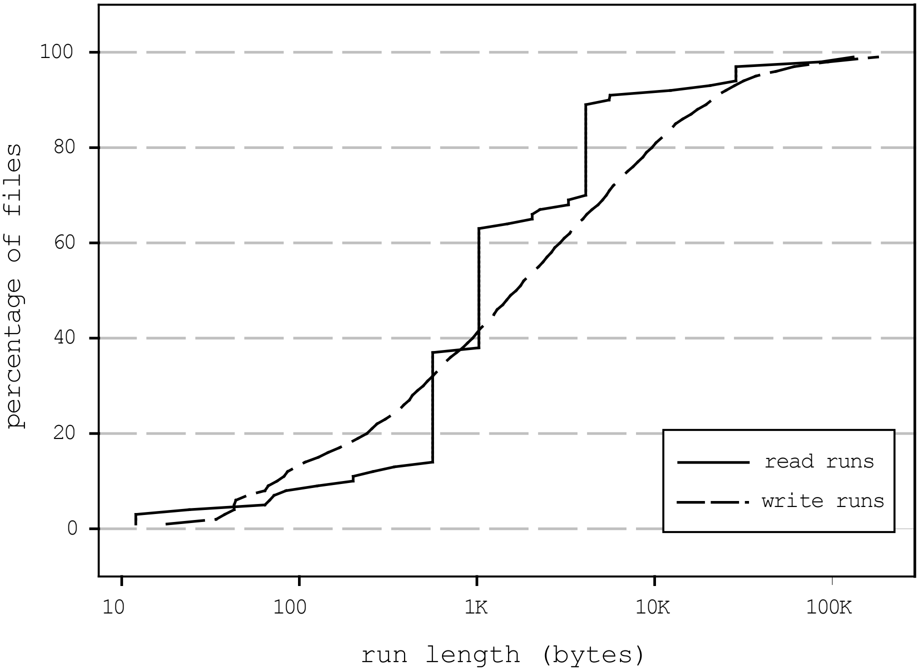


Figure 9.1. The cumulative distribution of the sequential run length weighted by the number of files

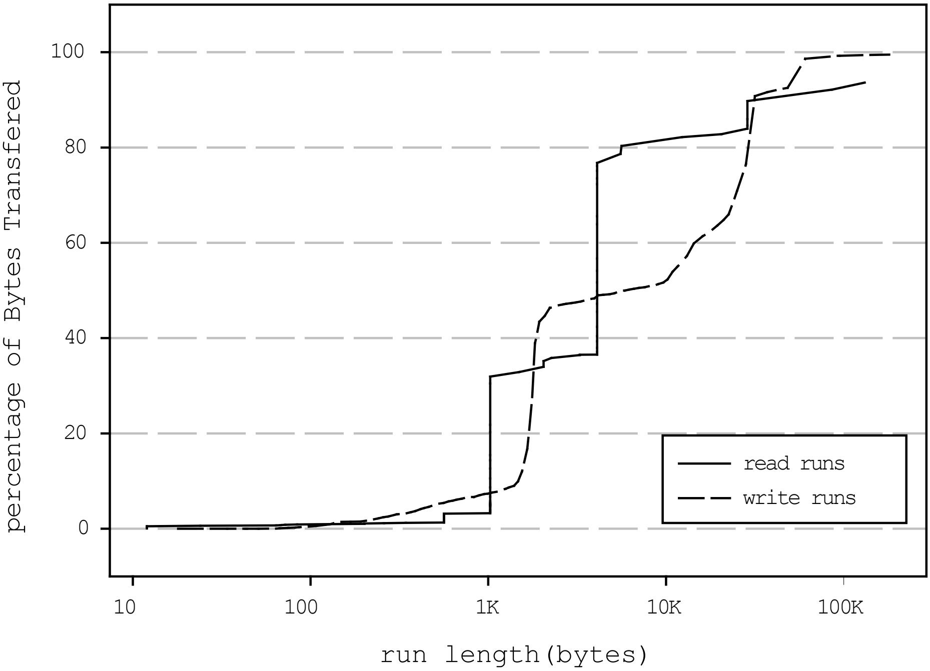


Figure 9.2. The cumulative distribution of the sequential run length weighted by bytes transferred

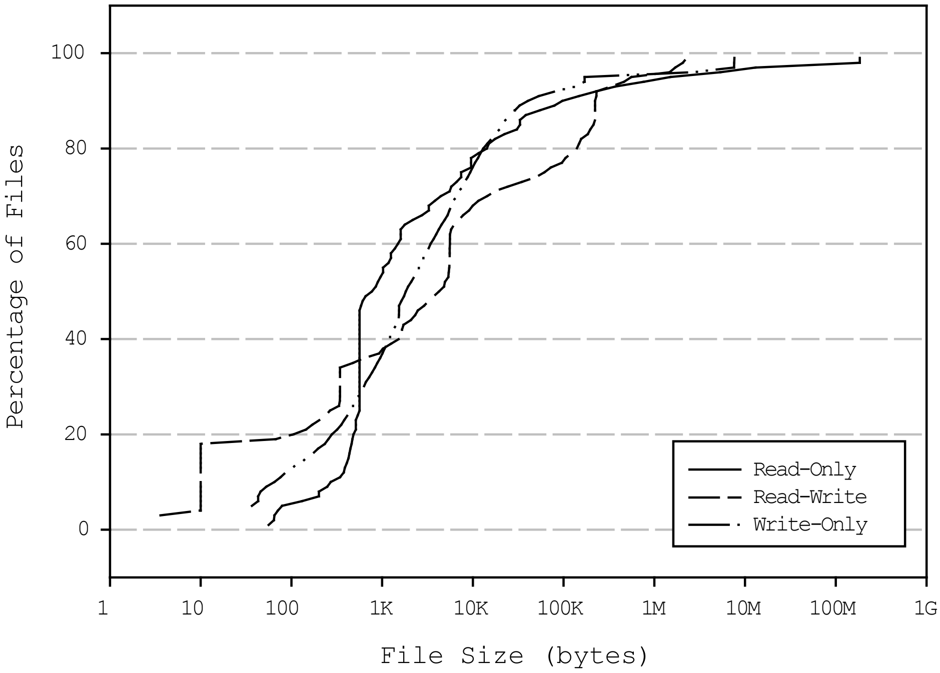


Figure 9.3. The filesize cumulative distribution, weighted by the number of files opened

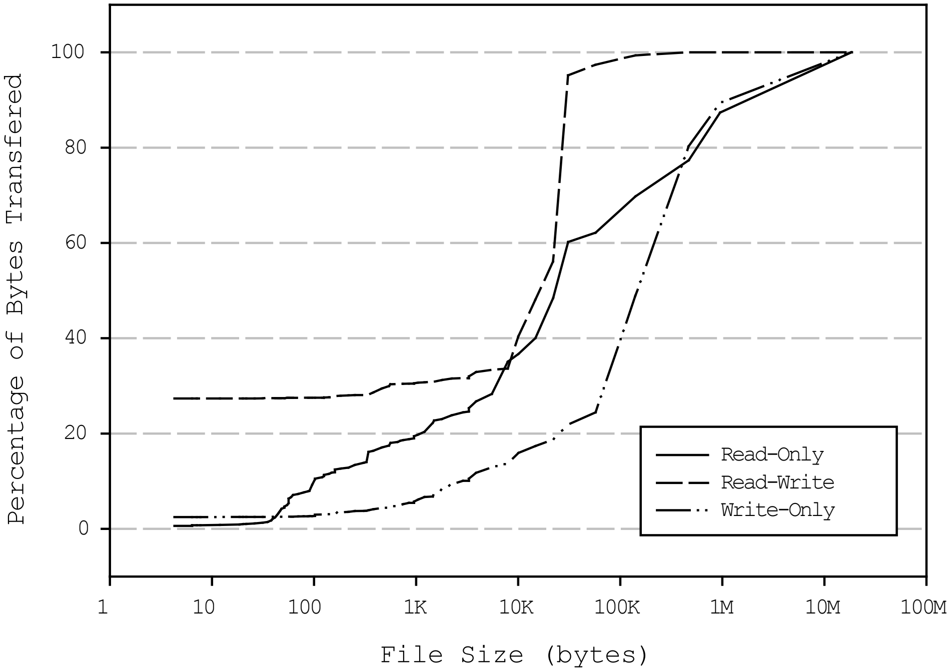


Figure 9.4. The filesize cumulative distribution, weighted by the number of bytes transfered

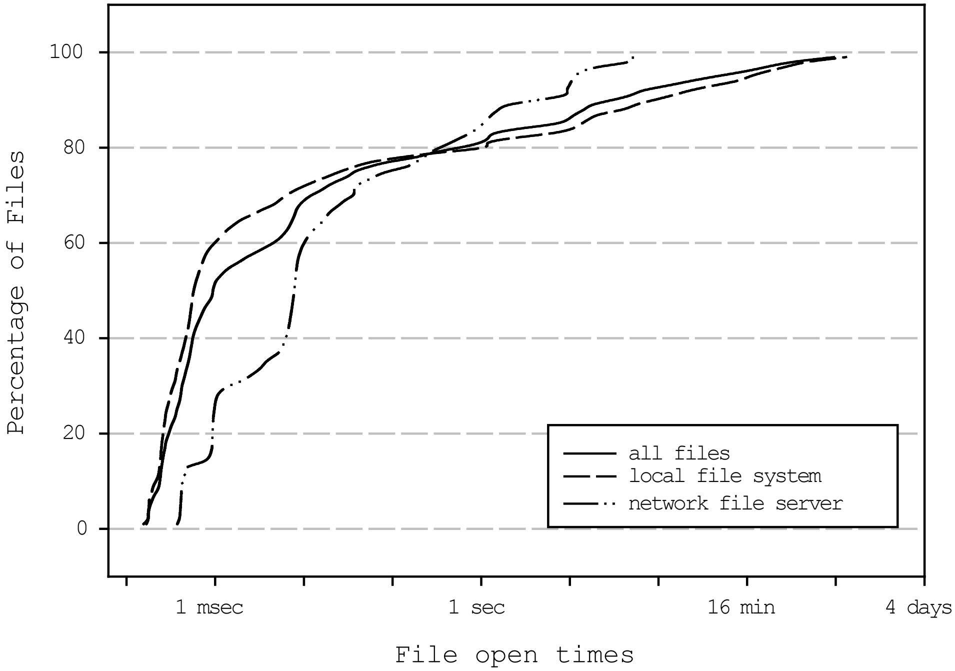


Figure 9.5. The file open time cumulative distribution, weighted by the number of files.

data was transferred sequentially versus 58% in the Windows NT traces. When comparing this type of trace summary there is a stronger presence of random access to data both in number of accesses and in the amount of data accessed for all file usage categories.

Another important access pattern examined is that of the *sequential runs*, which is when a portion of a file is read or written in a sequential manner. The prediction of a series of sequential accesses is important for effective caching strategies. When examining these runs we see that they remain short; the 80% mark for Sprite was below the 10 Kbytes, while in our traces we see a slight increase in run length with the 80% mark at 11 Kbytes (figure 9.1).

An observation about the Sprite traces was that most bytes were transferred in the longer sequential runs. The Windows NT traces support this observation, although the correlation is less prominent (figure 9.2). Access to large files shows increasing random access patterns, causing 15%-35% (in some traces up to 70%) of the bytes to be transferred in non-sequential manner.

If we look at all file open sessions for which data transfers where logged, not just those with sequential runs, we see that the 80% mark for the number of accesses changes to 24 Kbytes. 10% of the total transferred bytes were transferred in sessions that accessed at least 120 Kbytes.

When examining the size of files in relation to the number of sequential IO operations posted to them we see a similar pattern: most operations are to short files (40% to files shorter than 2K) while most bytes are transferred to large files (figures 9.3 and 9.4).

In Sprite the researchers found that, when examining the top 20% of file sizes, an increase of an order of magnitude was seen with respect to the BSD traces. This trend has continued: in the Windows NT traces the top 20% of files are larger than 4 Mbytes. An important contribution to this trend comes from the executables and dynamic loadable libraries in the distribution, which account for the majority of large files.

The last access pattern for which we examine the traces concerns the period of time during which a file is open. In this section we only look at file open sessions that have data transfer associated with them; sessions specific for control operation are examined in section 9.8. The results are presented in figure 9.5; about 75% of the files are open less than 10 milliseconds. This is a significant change when compared to the Sprite and BSD traces, which respectively measured a quarter-second and a half-second at 75%. The less significant difference between the two older traces was attributed to the fact that in the BSD traces the I/O was to local storage while in the Sprite the storage was accessed over the network. In the Windows NT traces we are able to examine these access times separately, and we found no significant difference in the access times between local and remote storage.

9.6.3 File lifetimes

The third measurement category presented in the BSD & Sprite traces is that of the lifetime of newly created files. The Sprite traces showed that between 65% and 80% of the new files were deleted within 30 seconds after they were created. In the Windows NT traces we see that the presence of this behavior is even stronger; up to 80% of the newly created files are deleted within 4 seconds of their creation.

In Windows NT we consider three sources for deletion of new files: (1) an existing file is truncated on open by use of a special option (37% of the delete cases), (2) a file is newly created or truncated and deleted using a delete control operation (62%), and (3) a file is opened with the *temporary file* attribute (1%).

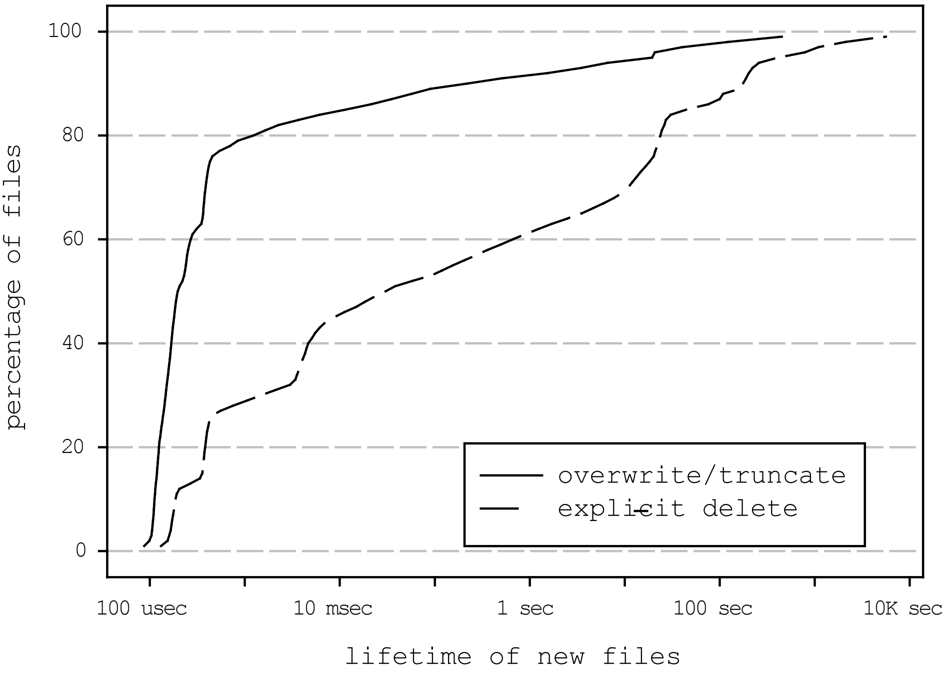


Figure 9.6. The lifetime of newly created filesgrouped by deletion method

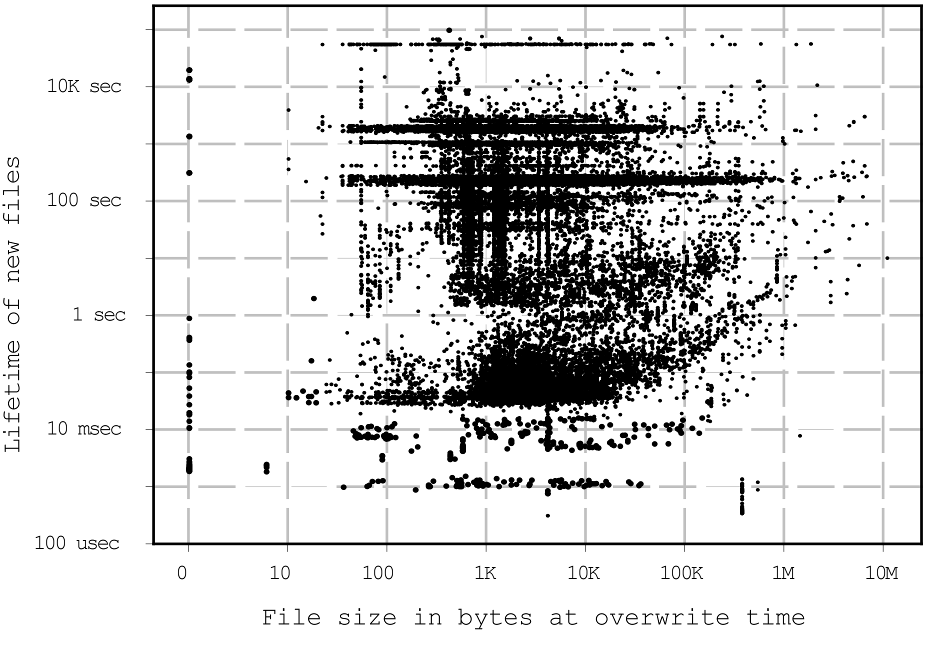


Figure 9.7. When examining file sizes at overwrite time, we cannot find a correlation between filesize and lifetime

In about 75% of the delete-through-truncate cases a file was overwritten within 4 milliseconds after it was created. The distribution shows a strong heavy tail with the top 10% having a lifetime of at least 1 minute, and up to 18 hours. If we inspect the time between the closing of a file and the subsequent overwrite action, we see that over 75% of these files are overwritten within 0.7 millisecond of the close.

In the case of explicitly deleted files, we see a higher latency between create and delete action. 72% of these files are deleted within 4 seconds after they were created and 60% 1.5 seconds after they were closed (see figure 9.6).

One of the possible factors in the difference in latency is related to which process deletes the file. In 94% of the overwrite cases, the process that overwrites the file also created it in the first place, while in 36% of the DeleteFile cases the same process deletes the file. A second factor is that there are no other actions posted to overwritten files, while in 18% of the DeleteFile cases, the file is opened one or more times between creation and deletion.

The *temporary file* attribute not only causes the file to be deleted at close time, but also prevents the cache manager's lazy writer threads from marking the pages containing the file data for writing to disk. Although it is impossible to extract the exact persistency requirements for temporary file usage from the traces, analysis suggests that at least 25%-35% of all the deleted new files could have benefited from the use of this attribute.

In 23% of the cases where a file was overwritten, unwritten pages were still present in the file cache when the overwrite request arrived. In the case of the CreateFile/ DeleteFile sequence 5% of the newly created files had still unwritten data present in the cache when deleted. Anomalous behavior was seen in 3% of the cases where the file was flushed from the cache by the application before it was deleted.

The apparent correlation between the file size and lifetime, as noticed by the Sprite researchers, is tremendously skewed by the presence of large files. In the Windows NT case only 4% of the deleted files are over 40 Kbytes and 65% of the files are smaller than 100 bytes. In the traces we could not find any proof that large temporary files have a longer lifetime. Figure 9.7 shows a plot of lifetime versus size of a trace sample, and although there are no large files in this plot that are deleted in less then 1 second, there is no statistical justification for a correlation between size and lifetime of temporary files.

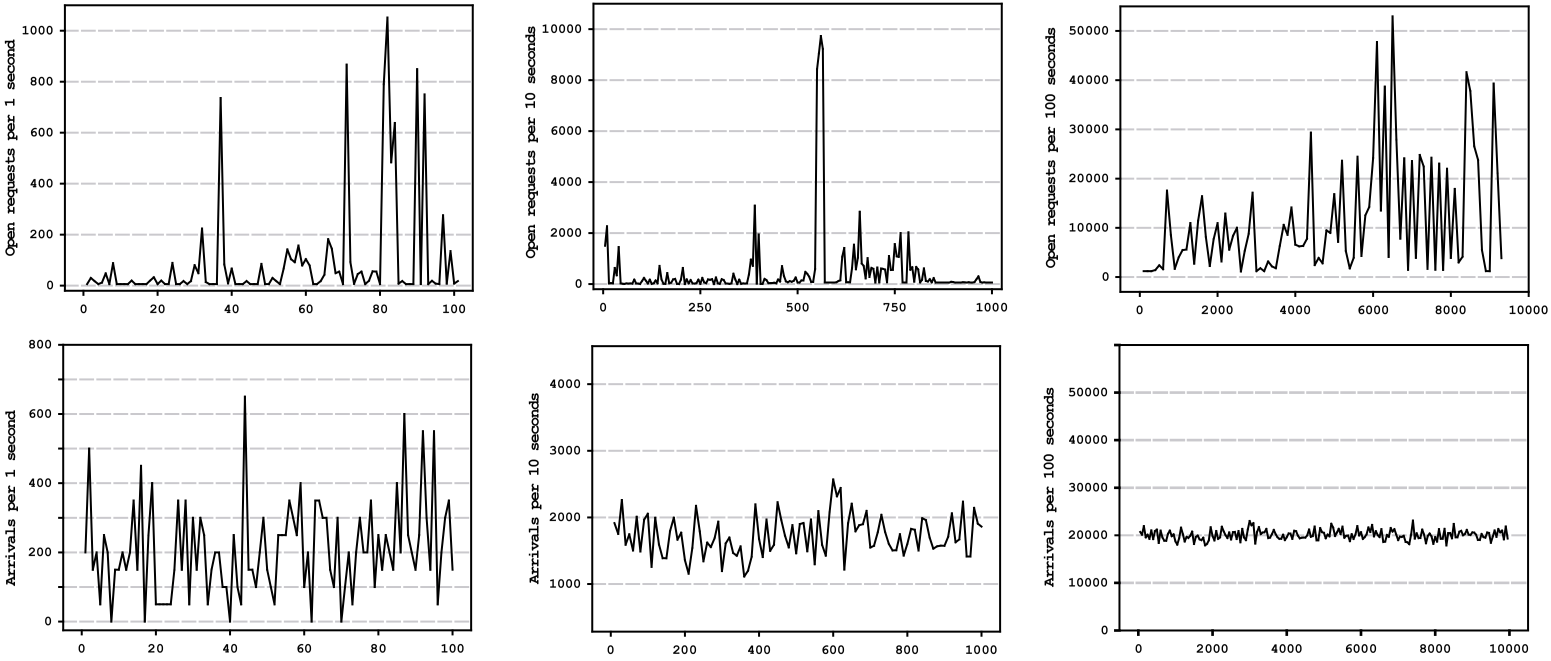


Figure 9.8. The top row displays the inter-arrival distribution of file open events at 3 different orders of magnitude. The bottom row contains a synthesized sample of a Poisson process with paramters estimated from the sample

9.7 Data distribution

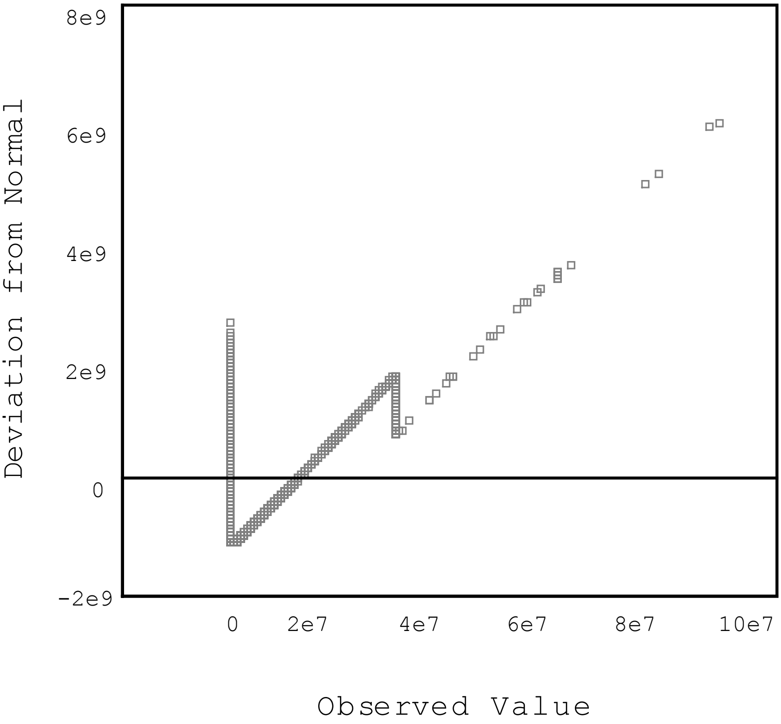
When analyzing the user activity in section 9.6.1 we were tempted to conclude that for Windows NT the average throughput in general has increased, but that the average in burst load has been reduced. The use of simple averaging techniques allows us to draw such conclusions, in similar fashion one could conclude from the file access patterns that most file accesses still occur in a read-only, whole-file sequential manner. If we examine the result of analysis of the file access in table 9.3 once more, the truly important numbers in that table are the ranges of values that were found for each of the statistical descriptives. The -/+ columns in the table represent the min/max values found when analyzing each trace separately.

When we have a closer look at the trace data and the statistical analysis of it, we find a significant variance in almost all variables that we can test. A common approach to statistically control burstiness, which is often the cause of the extreme variances, is to examine the data on various time scales. For example, in the previous two file system trace reports, the data was summarized over 10-second and 10-minute intervals, with the assumption that the 10-minute interval would smoothen any variances found in the traces.

If, for example, we consider the arrival rate of file system requests to be drawn from a Poisson distribution, we should see that the variances should diminish when we view the distribution at coarser time granularity. In figure 9.8 we compare the request arrival rates in one of our trace files, randomly chosen, with a synthesized sample from a Poisson distribution for which we estimated its mean and variance from the trace information (the details of this test are presented in [106]). When we view the samples at time scales with different orders of magnitude, we see that at larger time scales the Poisson sample becomes smooth, while the arrival data in our sample distribution continues to exhibit the variant behavior.

In almost all earlier file system trace research there is some notion of the impact of large files on the statistical analysis. In the Sprite research, for example, an attempt was made to discard the impact of certain categories of large files, by removing kernel development files from the traces. The result, however, did not remove the impact of large files, leading the researchers to conclude that the presence of large files was not accidental.

Analyzing our traces for the impact of outliers we find they are present in all categories. For example if we take the distribution of bytes read per open-close



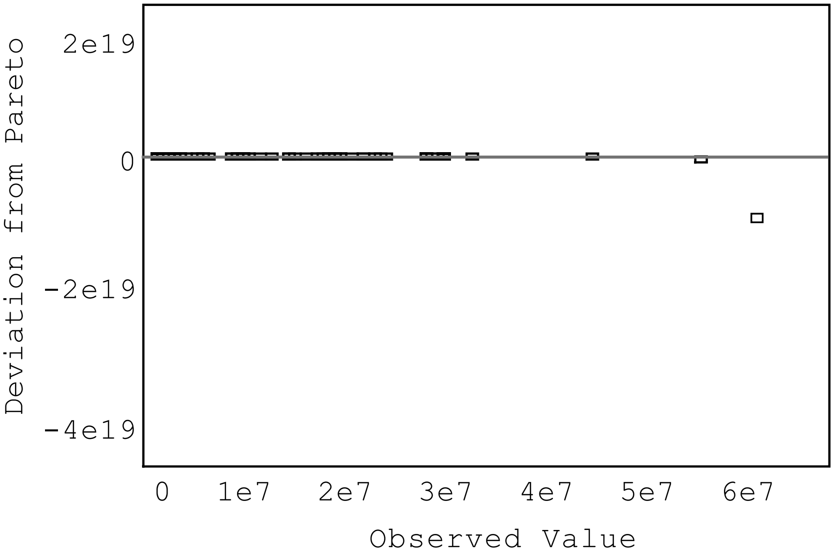


Figure 9.9. The arrival data from the sample used in figure 8 is tested against a Normal and a Pareto distribution through a QQ plot. The plot tests to what extend sample data follows a given distribution with estimated parameters.

session, we see that the mean of the distribution is forced beyond the 90th percentile by the impact of large file read sessions. If we visually examine how the sample distribution from figure 9.8 departs from normality through a QQ plot (figure 9.9) we see that values in the quartiles support the evidence that the distribution is not normal. If we use a QQ plot to test the sample against a Pareto distribution, which is the simplest distribution that can be used to model heavy-tail behavior, we see an almost perfect match.

To examine the tail in our sample distribution we produced a log-log complementary distribution plot (figure 9.10). The linear appearance of the plot is evidence of the power-law behavior of the distribution tail; normal or log-normal distributions would have shown a strong drop-off appearance in the plot. When we use a least-squares regression of points in the plotted tail to estimate the heavy-tail α parameter[1], we find a value of 1.2. This value is consistent with our earlier observation of infinite variance; however, we cannot conclude that the distribution also has an infinite mean [84].

This observation of extreme variance at all time scales has significant importance for operating system engineering and tuning: Resource predictions are often made based on the observed mean and variance of resource requests, assuming that, over time, this will produce a stable system. Evidence from our traces shows that modeling the arrival rates of I/O request as a Poisson process or size distributions as a Normal distribution is incorrect. Using these simplified assumptions can lead to erroneous design and tuning decisions when systems are not prepared for extreme variance in input parameters, nor for the long-range dependence of system events.

An important reason for the departure from a normal distribution in file system analysis is that user behavior has a very reduced influence on most of the file system operations. Whether it is file size, file open times, inter-arrival rates of write operations, or directory poll operations, all of these are controlled through loops in the applications, through application defined overhead to user storage, or are based on input parameters outside of the user's direct control. More than 92% of the file accesses in our traces were from processes that take no direct user input, even though

[1] A random variable X follows a heavy-tailed distribution if $P[X > x] \sim x^{-\alpha}$, as $x \to \infty$, $0 < \alpha < 2$. A reliable estimator for α is the *Hill* estimator. We have computed this for our samples and it confirms the more llcd plot estimation results. A value of $\alpha < 2$ indicates infinite variance, if $\alpha < 1$ this also indicates an infinite mean.

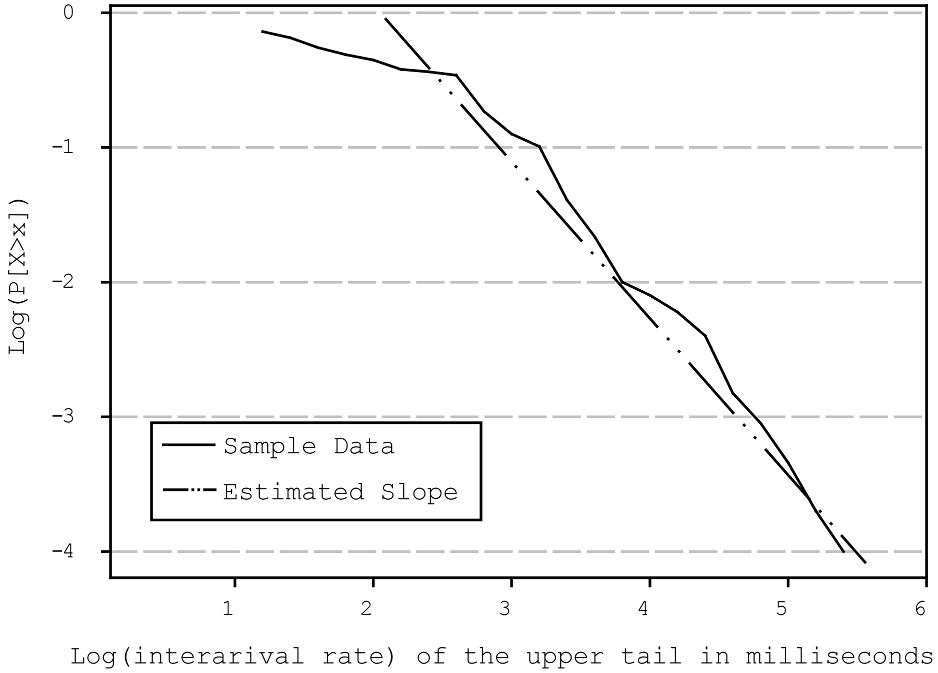


Figure 9.10. A log-log complementary distribution plot for the tail of the sample from figure 8, combined with a fitted line for the estimation of the α parameter

all the systems were used interactively. From those processes that do take user input, *explorer.exe*, the graphical user interface, is dominant, and although the user controls some of its operation, it is the structure and content of the file system that determines explorer's file system interactions, not the user requests. Unfortunately this kind of information cannot be extracted form the older traces so we cannot put this into a historical perspective.

This process controlled dominance of file system operations is similar to observations in data-communication, where, for example, the length of TCP sessions are process controlled, with only limited human factors involved. File system characteristics have an important impact on the network traffic; as for example the file size is a dominant factor in WWW session length. Given that the files and directories have heavy-tailed size distributions, this directly results into heavy-tailed distributions for those activities that depend on file system parameters [21,45].

Another important observation is that some characteristics of process activity, independent of the file system parameters, also play an important role in producing the heavy-tailed access characteristics. From the analysis of our traces we find that process lifetime, the number of dynamic loadable libraries accessed, the number of files open per process, and spacing of file accesses, all obey the characteristics of heavy-tail distributions. Some of these process characteristics cannot be seen as completely independent of the file system parameters; for example, the lifetime of the *winlogon* process is determined by the number and size of files in the user's profile.

Our observations of heavy-tail distributions in all areas of file system analysis lead to the following general conclusions:

1. We need to be very careful in describing file system characteristics using simple parameters such as average and variance, as they do not accurately describe the process of file system access. At minimum we need to describe results at different granularities and examine the data for extreme variances.
2. In our design of systems we need to be prepared for the heavy-tail characteristics of the access patterns. This is particularly important for the design and tuning of limited resource systems such as file caches, as there is important evidence that heavy-tail session length (such as open times and amount of bytes transferred) can easily lead to queue overflow and memory starvation [48].
3. When constructing synthetic workloads for use in file system design and benchmarking we need to ensure that the infinite variance characteristics are properly modeled in the file system test patterns. In [92], Seltzer et al. argue for application-specific file system benchmarking, which already allows more focused testing, but for each test application we need to ensure that the input parameters from the file system under test and the ON/OFF activity pattern of the application is modeled after the correct (heavy-tailed) distributions.
4. When using heuristics to model computer system operations it is of the highest importance to examine distributions for possible self-similar properties, which indicate high variance. Exploitation of these properties can lead to important improvements in the design of systems, as shown in [47].

9.8 Operational characteristics

There were 3 focus points when we analyzed the traces to understand the specifics of the Windows NT file system usage:

- Examine the traces from a system engineering perspective: the arrival rate of events, the holding time of resources, and the resource requests in general.

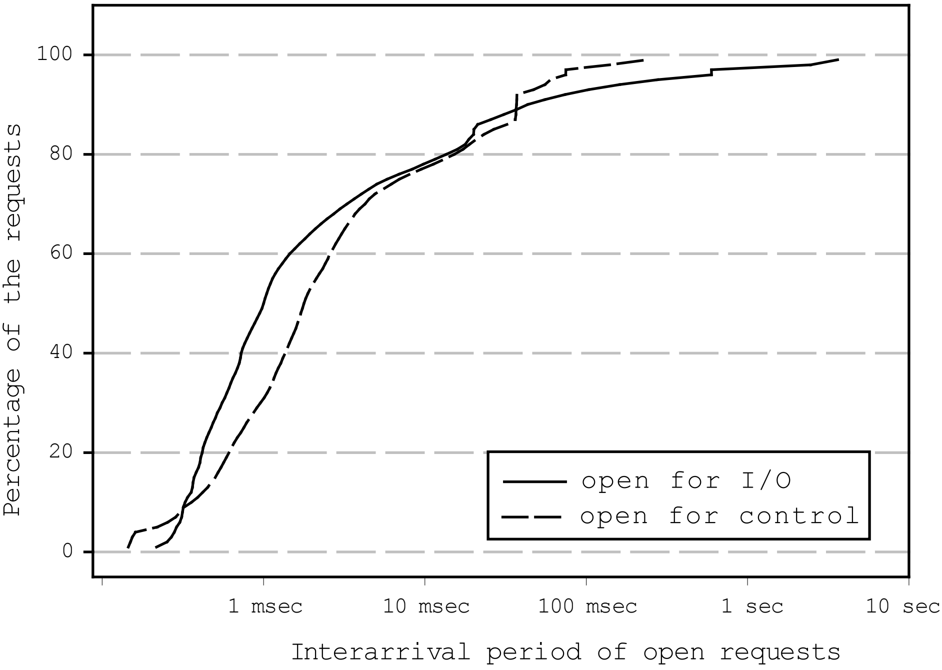


Figure 11. Cumulative distributionof the inter-arrival periods of the files system open requests, per usage type

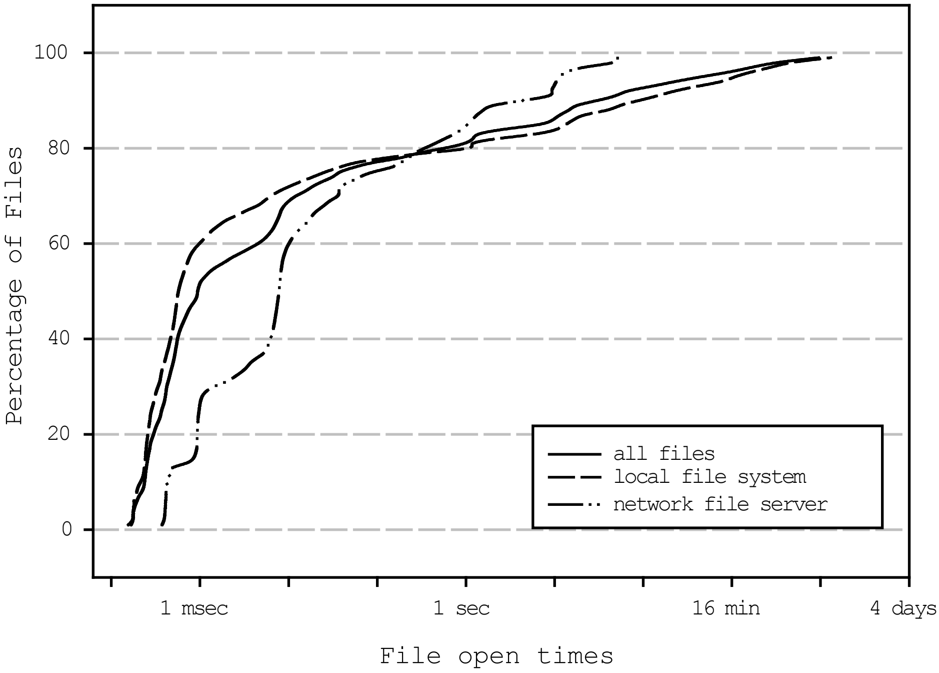


Figure 9.12. Cumulative distributionof the periods that files are open per usage type.

- Gain understanding in how applications use the functionality offered through the broad Windows NT file system interface and how the various options are exploited.
- Investigate the complexity of the Windows NT application and file system interactions.

In this section we explore these points by looking at the different file system operations, while in the next 2 sections we will investigate cache manager related technologies from these perspectives.

9.8.1 Open and close characteristics

Any sequence of operations on a file in Windows NT is encapsulated in an Open/ Close sequence of events. Some operating systems have core primitives such as rename and delete which do not require the caller to open the file first, but in Windows NT these operations are generic file operations on files that have been opened first. For example, the deletion of a file or the loading of an executable can only be performed after the file itself has been opened.

Figure 9.11 displays inter-arrival times of open requests arriving at the file system: 40% of the requests arrive within 1 millisecond of a previous request, while 90% arrives with 30 milliseconds. When we investigate the arrivals by grouping them into intervals, we see that only up to 24% of the 1-second intervals of a user's session have open requests recorded for them. This again shows us the extreme burstiness of the system.

If we examine the reuse of files, we see that between 24% and 40% of the files that are opened read-only are opened multiple times during a user's session. Of the files accessed write-only, 4% are opened for another write-only session, while 36%-52% are re-opened for reading. 94% of the files that were open for reading and writing are opened multiple times, in the same mode.

An important measurement for resource tuning is the time that files are kept open (file session lifetime). In figure 9.12 we present the session lifetimes for a number of cases. The overall statistics show that 40% of the files are closed within one millisecond after they were opened and that 90% are open less then one second. Of the sessions with only control or directory operations 90% closed within 10 milliseconds.

When we investigate session times for the type of data access, we see that 70% of read-write access happens in periods of less then 1 second, while read-only and write-only accesses have this 1 second mark at 60% and 30%, respectively.

The session length can also be viewed from the process perspective. Some processes only have a single style of file access and the session time for each access is similar. The FrontPage HTML editor, for example, never keeps files open for longer then a few milliseconds. Others such as the development environments, databases control engines or the services control program keep 40%-50% of their files open for the complete duration of their lifetime. Programs such as *loadwc*, which manages a user's web subscription content, keep a large number of files open for the duration of the complete user session, which may be days or weeks. The first approach, opening a file only for the time necessary to complete IO, would produce a correlation between session time and file size. When testing our samples for such a correlation we could not find any evidence.

In general it is difficult to predict when a file is opened what the expected session time will be. All session distributions, however, had strong heavy-tails, from which we can conclude that once a file is open for a relatively long period (3-5 seconds, in most cases) the probability that the file will remain open for a very long time is significant.

Windows NT has a two stage close operation. At the close of the file handle by the process or kernel module, the IO manager sends a *cleanup* request down the chain of drivers, asking each driver to release all resources. In the case of a cached file, the cache manager and the VM manager still hold references to the FileObject, and the cleanup request is a signal for each manager to start releasing related resources. After the reference count reaches zero, the IO manager sends the *close* request to the drivers. In the case of read caching this happens immediately as we see the close request within 4-8 µsec after the cleanup request. In the case of write caching the references on the FileObject are released as soon as all the dirty pages have been written to disk, which may take 1-4 seconds.

9.8.2 Read and write characteristics

The burst behavior we saw at the level of file open requests has an even stronger presence at the level of the read and write requests. In 70% of the file opens, read/write actions were performed in batch form, and the file was closed again. Even in the case of files that are open longer than the read/write operations require, we see that the reads and writes to a file are clustered into sets of updates. In almost 80% of the reads, if the read was not at the end-of-file, a follow-up read will occur within 90 microseconds. Writes occur at an even faster pace: 80% have an inter-arrival space of less than 30 microseconds. The difference between read and write intervals is probably related to the fact that the application performs some processing after each read, while the writes are often pre-processed and written out in batch style.

When we examine the requests for the amount of data to be read or written, we find a distinct difference between the read and write requests. In 59% of the read cases the request size is either 512 or 4096 bytes. Some of the common sizes are triggered by buffered file i/o of the *stdio* library. Of the remaining sizes, there is a strong preference for very small (2-8 bytes) and very large (48 Kbytes and higher) reads. The write sizes distribution is more diverse, especially in the lower bytes range (less then 1024 bytes), probably reflecting the writing of single data-structures.

9.8.3 Directory & control operations

The majority of file open requests are not made to read or write data. In 74%, the open session was established to perform a directory or a file control operation.

There are 33 major control operations on files available in Windows NT, with many operations having subdivisions using minor control codes. Most frequently used are the major control operations that test whether path, names, volumes and objects are valid. In general the application developer never requests these operations explicitly, but they are triggered by the Win32 runtime libraries. For example, a frequently arriving control operation is whether the "*volume is mounted*", which is issued in the name verification part of directory operations. This control operation is issued between up to 40 times a second on any reasonably active system.

Another frequently issued control operation is *SetEndOfFile*, which truncates the file to a given size. The cache manager always issues it before a file is closed that had data written to it. This is necessary as the delayed writes through the VM manager always have the size of one or more pages, and the last write to a page may write more data than there is in the file. The end-of-file operation then moves the end-of-file mark back to the correct position.

9.8.4 Errors

Not all operations are successful: of the open requests 12% fail and of the control operations 8% fail. In the open cases there are two major categories of errors: the file to be opened did not exist in 52% of the error cases and in 31% the creation of a file was requested, but it already did exist. When we examine the error cases more closely we see that a certain category of applications that uses the “open” request as a test for the existence of the file: the failure is immediately followed by a create action, which will be successful.

Reads hardly ever fail (0.2%); the error that does occur on the read are attempts to read past the end-of–file. We did not find any write errors.

9.9 The cache manager

An important aspect of the Windows NT file system design is the interaction with the cache manager. The Windows NT kernel is designed to be extensible with many third party software modules, including file systems, which forces the cache manager to provide generalized support for file caching. It also requires file system designers to be intimately familiar with the various interaction patterns between file system implementation, cache manager and virtual memory manager. A reasonably complete introduction can be found in [67].

In this section we will investigate two file system and cache manager interaction patterns: the read-ahead and lazy-write strategies for optimizing file caching. The cache manager never directly requests a file system to read or write data; it does this implicitly through the Virtual Memory system by creating memory-mapped sections of the files. Caching takes place at the logical file block level, not at the level of disk blocks.

A process can disable read caching for the file at file open time. This option is hardly ever used: read caching is disabled in only 0.2% of all files that had read/write actions performed on them. 76% of those files were data files from opened by the “system” process. All of these files were used in a read-write pattern with a *write-through* option set to also disable write caching. Developers using this option need to be aware of the block size and alignment requirements of the underlying file system. All of the requests for these files will go through the traditional IRP path.

9.9.1 Read-ahead

When caching is initialized for a file, the Windows NT cache manager tries to predict application behavior and to initiate file system reads before the application requests the data, in order to improve cache hit rate. The standard granularity for read-ahead operation is 4096 bytes, but is under the control of the file system, which can change it on a per file basis. In many cases the FAT and NTFS file systems boost the read-ahead size to 65 Kbytes. Caching of a file is initiated when the first read or write request arrives at the file system driver.

Of all the sessions that performed reads 31% used a single IO operation to achieve their goal, and although this caused the caching to be initiated and data to be loaded in the cache, the cached data was never accessed after the first read.

Of the sequential accesses with multiple reads, which benefit from the read-ahead strategy, 40% used read sizes smaller than 4Kbytes and 92% smaller than 65Kbytes. This resulted in that only 8% of the read sequences required more than a single read-ahead action.

The cache manager tries to predict sequential access to a file so it can load data even more aggressively. If the application has specified at open time that the file data will be processed through sequential access only, the cache manager doubles the size of the read-ahead requests. Of file-opens with sequential read accesses only 5% specified this option. Of those files 99% were smaller than the read-ahead granularity and 80% smaller than a single page, so the option has no effect.

The cache manager also tries to predict sequential access by tracking the application actions: read-ahead is performed when the 3rd of a sequence of sequential requests arrives. In our traces this happened in 7% of the sequential cases that needed data beyond the initial read-ahead.

The cache manager uses a fuzzy notion of sequential access; when comparing requests, it masks the lowest 7 bits to allow some small gaps in the sequences. In our test in section 9.6.2, this would have increased the sequential marked trace runs by 1.5%.

9.9.2 Write-behind

Unless explicitly instructed by the application, the cache manager does not immediately write new data to disk. A number of lazy-write worker threads perform a scan of the cache every second, initiating the write to disk of a portion of the dirty pages, and requesting the close of a file after all references to the file object are released. The algorithm for the lazy-writing is complex and adaptive, and is outside of the scope of this description. What is important to us is the bursts of write requests triggered by activity of the lazy-writer threads. In general, when the bursts occur, they are in groups of 2-8 requests, with sizes of one or more pages up to 65 Kbytes.

Applications have two methods for control over the write behavior of the cache. They can disable write caching at file open time, or they can request the cache manager to write its dirty pages to disk using a flush operation.

In 1.4% of file opens that had write operations posted to them, caching was disabled at open time. Of the files that were opened with write caching enabled, 4% actively controlled their caching by using the flush requests. The dominant strategy used by 87% of those applications was to flush after each write operation, which suggests they could have been more effective by disabling write caching at open time.

9.10 FastIO

For a long time the second access path over which requests arrived at the file system driver, dubbed the *FastIO* path, has been an undocumented part of the Windows NT kernel. The Device Driver Kit (DDK) documentation contains no references to this part of driver development, which is essential for the construction of file systems. The Installable File System Kit (IFS) shipped as Microsoft's official support for file system development, contains no documentation at all. Two recent books [67,96] provide some insight into the role of the FastIO path, but appear unaware of its key role in daily operations. In this section we will examine the importance of this access path, and provide some insight into its usage.

For some time the popular belief, triggered by the unwillingness of Microsoft to document FastIO, was that this path was a private "hack" of the Windows NT kernel developers to secretly bypass the general IO manager controlled IRP path. Although FastIO is a procedural interface, faster when compared with the message-passing interface of the IO manager, it is not an obscure hack. The "fast" in FastIO does not refer to the access path but to the fact that the routines provide a direct data path to the cache manager interface as used by the file systems. When file system drivers indicate that caching has been initialized for a file, the IO manager will try to transfer the data directly in and out of the cache by invoking methods from the FastIO interface. The IO manager does not invoke the cache manager directly but first allows file system filters and drivers to manipulate the request. If the request does not return a success value, the IO manager will in most cases retry the operation over the traditional IRP path. File system filter drivers that do not implement all of methods of the FastIO interface, not even as a passthrough operation, severely handicap the system by blocking the access of the IO manager to the FastIO interface of the underlying file system and thus to the cache manager.

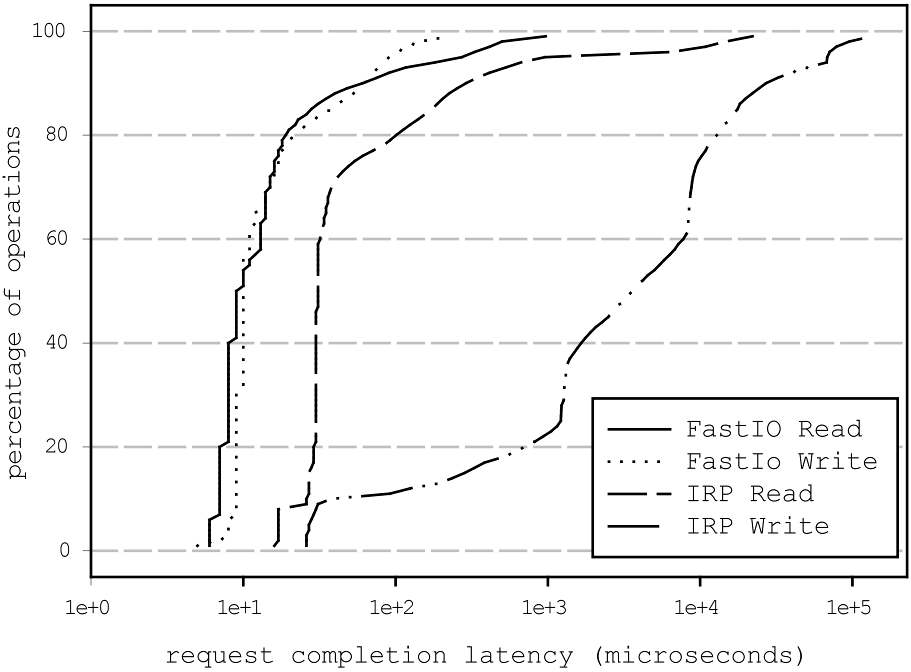


Figure 9.13. The cumulative distribution of the service period for each of the 4 major request types.

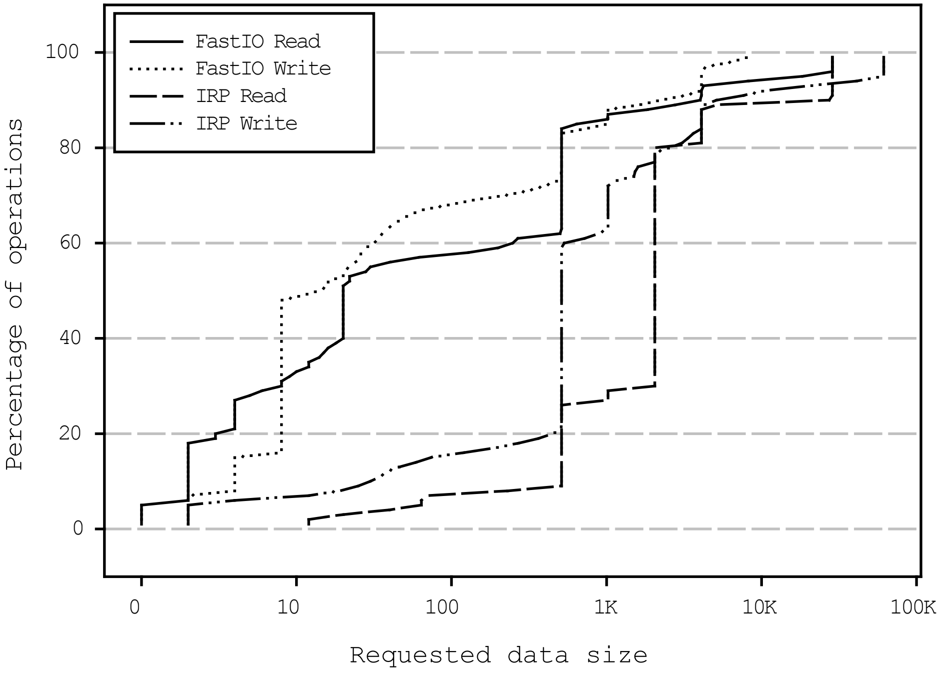


Figure 9.14. The cumulative distribution of the data request size for each of the 4 major request types

Caching is not performed automatically for each file; a file system has to explicitly initialize caching for each individual file and in general a file system delays this until the first read or write request arrives. This results in a file access pattern where the traces will log a single read or write operation through the IRP interface, which sets up caching for that file, followed by a sequence of FastIO calls that interact with the file cache directly. The effect on latency of the different operations is shown in figure 9.13.

If we examine the size of the read requests in figure 9.14, we see that FastIO requests have a tendency towards smaller size. This is not related to the operation itself, but to the observation that processes that use multiple operations to read data, in general use more targeted sized buffers to achieve their goal. Processes that use only a few operations do this using larger buffers (page size, 4096 bytes, being the most popular).

Some processes takes this to the extreme; a non-Microsoft mailer uses a single 4Mbyte buffer to write to its files, while some of the Microsoft Java Tools read files in 2 and 4 byte sequences, often resulting in thousands of reads for a single class file.

The cache manager has functionality to avoid a copy of the data through a direct memory interface, providing improved read and write performance, and this functionality can be accessed through the IRP as well as the FastIO interface. We observed that only kernel-based services use this functionality.

9.11 Related work

File tracing has been an important tool for designing file systems and caches. There are 3 major tracing studies of general file systems: the BSD and Sprite studies [5,71], which were closely related and examined an academic environment. The 3rd study examined in detail the file usage under VMS at a number of commercial sites [78]. One of our goals was to examine the Windows NT traces from an operating system perspective; as such we compared our results with those found in the BSD and Sprite studies. The VMS study focused more on the differences between the various usage types encountered, and a comparison with our traces, although certainly interesting, was outside of the scope of this research.

A number of other trace studies have been reported, however, they either focused on a specific target set, such as mobile users, or their results overlapped with the 3 major studies [25,59,66,107].

There is a significant body of work that focuses on specific subsets of file system usage, such as effective caching, or file system and storage system interaction.

There have been no previous reports on the tracing of file systems under Windows NT. A recent publication from researchers at Microsoft Research examines the content of Windows NT file systems, but does not report on trace-based usage [28].

With respect to our observations of heavy-tails in the distributions of our trace data samples; there is ample literature on this phenomenon, but little with respect to operating systems research. A related area with recent studies is that of wide-area network traffic modeling and World Wide Web service models.

In [45], Gribble, et al. inspected a number of older traces, including the Sprite traces, for evidence of self-similarity and did indeed find such evidence for short, but not for long term behavior. They did conclude that the lack of detail in the older traces made the analysis very hard. The level of detail of the Windows NT traces is sufficient for this kind of analysis.

9.12 Summary

To examine file system usage we instrumented a collection of Windows NT 4.0 systems and traced, in detail, the interaction between processes and the file system. We compared the results of the traces with the results of the BSD and Sprite studies [5,71] performed in 1985 and 1991. A summary of our observations is presented in table 9.1.

We examined the samples for presence of heavy-tails in the distributions and for evidence of extreme variance. Our study confirmed the findings of others who examined smaller subsets of files: that files have a heavy-tail size distribution. But more importantly we encountered heavy-tails for almost all variables in our trace set: session inter-arrival time, session holding times, read/write frequencies, read/write buffer sizes, etc. This knowledge is of great importance to system engineering, tuning and benchmarking, and needs to be taken into account when designing systems that depend on distribution parameters.

When we examined the operational characteristics of the Windows NT file system we found further evidence of the extreme burstiness of the file systems events. We also saw that the complexity of the operation is mainly due to the large number of control operations issued and the interaction between the file systems, cache manager and virtual memory system.

The file system cache manager plays a crucial role in the overall file system operation. Because of the aggressive read-ahead and write-behind strategies, an amplification of the burstiness of file system requests occurs, this time triggered by the virtual memory system.

We examined the undocumented FastIO path and were able to shed light on its importance and its contribution to the overall Windows NT file system operation.

In this chapter we reported on the first round of analysis of the collected trace data. There are many aspects of file system usage in Windows NT that have not been examined such as file sharing, file locking, details of the control operations, details of the various file cache access mechanisms, per process and per file type access characteristics, etc. We expect to report on this in the future.

Bibliography

[1] Abbot, M., and Peterson, L., Increasing Network Throughput by Integrating Protocol Layers, *IEEE/ACM Transactions on Networking*, Vol. 1, No. 5, pages 600-610, Oct. 1993.

[2] Anderson, T.E., Culler, D.E., Patterson, D.A., and the NOW Team, A Case for NOW (Networks of Workstations), *IEEE Micro*, Feb. 1995, pages 54-64.

[3] Badovinatz, P., Chandra, T.D., Gopal, A., Jurgensen, D., Kirby, T., Krishnamur, S., and Pershing, J., GroupServices: infrastructure for highly available, clustered computing, unpublished document, December 1997.

[4] Bailey, M., Gopal, B., Pagels, M., Peterson, L., and. Sarkar, P., Pathfinder: A pattern Based Packet Classifier, in *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation*, pages 115-124, Nov. 1994.

[5] Baker, M.G., Hartmann, J.H., Kupfer, M.D., Shirriff, K.W. and Ousterhout, J.K., Measurement of a Distributed File System, in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198-212, Pacific Grove, CA, October 1991.

[6] Birman, K.P, The Process Group Approach to Reliable Distributed Computing, *Communications of the ACM*, vol. 36, no. 12, December 1993.

[7] Birman, K.P., and Renesse, R. van, Software for Reliable Networks, *Scientific American*, May, 1996.

[8] Birman, K.P., *Building Secure and Reliable Network Applications*. Manning Publishing Co. and Prentice Hall, 1997.

[9] Birman, K.P., Reliable Multicast Goes Mainstream, Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS) Spring 1998 (Volume 10, Number 1).

[10] Birman, K.P, Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M. and Minsky, Y., Bimodal Multicast. *ACM Transactions on Computer Systems*, vol. 17, no. 2, May 1999.

[11] Birman, K.P., A Review of Experiences with Reliable Multicast. Software *Practice and Experience*, vol. 9, 1999.

[12] Blumrich, M., Dubnicki, C., Felten, E. W., and Li, K., Virtual- Memory-Mapped Network Interfaces, *IEEE Micro*, Feb. 1995, pages 21-28.

[13] Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N. and Su, W., Myrinet: A Gigabit-per-Second Local Area Network, *IEEE Micro*, Feb. 1995, pp. 29-36.

[14] Brakmo, L., O'Malley, S. and Peterson, L. TCP Vegas: New Techniques for Congestion Detection and Avoidance, in *Proceedings of ACM SIGCOMM-94*, pages 24-35, Aug 1994.

[15] Braun, T., and Diot, C., Protocol Implementation Using Integrated Layer Processing, in *Proceedings of ACM SIGCOMM-95*, Sept 1995.

[16] Carr, R., Tandem Global Update Protocol, *Tandem Systems Review*, V1.2 1985.

[17] Chun, B.N., Mainwaring, A.M., and Culler, D.E., Virtual Network Transport Protocols for Myrinet, *IEEE Micro*, Jan.-Feb. 1998, pp. 53-63.

[18] Clark, D., and Tennenhouse, D., Architectural Considerations for a New Generation of protocols, in *Proceedings of ACM SIGCOMM-87*, pages 353-359, Aug. 1987.

[19] Clark, D., Jacobson, V., Romkey, J. and Salwen, H., An Analysis of TCP Processing Overhead, *IEEE Transactions on Communications*, pages 23-29, June 1989.

[20] Cluster Infrastructure for Linux, http://ci-linux.sourcefourge.net, December, 2001.

[21] Crovella, M., Taqqu, M., Bestevaros, A., Heavy-Tailed Probability Distributions in the World Wide Web, in *A Practical Guide to Heavy-Tails: Statistical Techniques and Applications*, R. Adler, R. Feldman and M.S. Taqqu, Editors, 1998, Birkhauser Verlag, Cambridge, MA.

[22] Culler, D. E., Dusseau, A., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., von Eicken, T., and Yelick, K., introduction to Split-C, in *Proceedings of Supercomputing `93*, 1993

[23] Culler, D. E., Dusseau, A., Martin, R., Schauser, K. E., Fast Parallel Sorting: from LogP to Split-C, in *Proceedings of WPPP`93*, July 93.

[24] Culler, D.E., Keeton, K., Krumbein, L., Liu, L.T., Mainwaring, A. R. Martin, R., Rodrigues, S., Wright, K., and Yoshikawa, C., Generic Active Message Interface Specification, version 1.1, February 1995, available at http://now.cs.berkeley.edu/Papers/Papers/gam_spec.ps.

[25] Dahlin, M., Mather, C., Wang, R., Anderson, T. and Patterson, D., A Quantitative Analysis of Cache Policies for Scalable Network File Systems, in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150 - 160, Nashville, TN, May 1994.

[26] Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A. and Lumley, J., Afterburner, *IEEE Network Magazine*, Vol 7, No. 4, pages 36-43, July 1993.

[27] Devlin, B.; Gray, J.; Laing, B.; Spix, G., Scalability Terminology: Farms, Clones, Partitions, and Packs: RACS and RAPS, Microsoft Research Technical Report, December, 1999.

[28] Douceur, John, and William Bolosky, A Large Scale Study of File-System Contents, in *Proceedings of the SIGMETRICS'99 International Conference on Measurement and Modeling of Computer Systems*, pages 59-70, Atlanta, GA, May 1999.

[29] Druschel, P., and Peterson, L., Fbufs: A High-Bandwidth Cross-Domain Transfer Facility, in *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 189-202. December 1993.

[30] Druschel, P., Peterson, L., and Davie, B.S., Experiences with a High-Speed Network Adaptor: A Software Perspective, in *Proceedings of ACM SIGCOMM-94*, pages 2-13, Aug 1994.

[31] Dubnicki, C., Bilas, A., Chen, Y., Damianakis, S.N., and Li. K., Myrinet Communication, *IEEE Micro*, Jan.-Feb. 1998, pp. 50-52.

[32] Dunning, D. and Regnier, G., The Virtual Interface Architecture, in *Proceedings of the Symposium on Hot Performance Interconnect Architectures*, Stanford University, 1997.

[33] Dunning, D., G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd, The Virtual Interface Architecture, *IEEE Micro*, Mar.-Apr. 1998, pp. 66-76.

[34] Edwards, A., Watson, G., Lumley, J., Banks, D., Calamvokis, C. and Dalton, C., User-space protocols deliver high performance to applications on a low-cost Gb/s LAN, in *Proceedings of ACM SIGCOMM-94*, pages 14-23, Aug. 1994.

[35] Eicken, T. von, Culler, D. E., Goldstein, S. C., and Schauser, K. E., Active Messages: A Mechanism for Integrated Communication and Computation, in *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256-266, May 1992.

[36] Eicken, T. von, Basu, A., and Buch, V., Low-Latency Communication over ATM Networks Using Active Messages, *IEEE Micro*, Feb. 1995, pages 46-53.

[37] Eicken, T von, Basu, A., Buch, V., and Vogels, W., U-Net: A User-Level network Interface for Parallel and Distributed Computing, in *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec 1995, pp 40-53.

[38] Eicken, T von, and Vogels, W., Evolution of the Virtual Interface Architecture, *IEEE Computer*, Nov 1998.

[39] Fineberg, S.A., and Mehra, P., The Record Breaking Terabyte Sort on a Compaq Cluster, in *Proceedings of the 3rd Usenix Windows NT Symposium*, Seattle, WA, July 1999.

[40] Flanigan, P. and Jawed Karim J., NCSA Symera Distributed parallel-processing using DCOM, *Dr. Dobb's Journal*, November 1998.

[41] Floyd, S., Jacobson, V., and McCanne, S., A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, in *Proceedings of ACM SIGCOMM-95*, Sept 1995.

[42] Fox, A., Gribble, S.D., Chawathe, Y., Brewer, E.A., Gauthier, P., Cluster-Based Scalable Network Services, in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Sant Malo, France, September 1997.

[43] Friedman R., and Birman, K.P., Using Group Communication Technology to Implement a Reliable and Scalable Distributed IN Coprocessor, in *Proceedings of TINA 96*, Heidelberg, Germany, 1996.

[44] Gray, J., and Reuter A., *Transaction Processing Concepts and Techniques*, Morgan Kaufmann, 1994.

[45] Gribble, Steven, Gurmeet SinghManku, Drew Roselli, Eric A.Brewer, Timothy J.Gibson, and Ethan L.Miller; Self-similarity in file systems , in *Proceedings of the SIGMETRICS'98 / PERFORMANCE'98 joint International Conference on Measurement and Modeling of Computer Systems*, pages 141 - 150, Madison, WI, June 1998.

[46] Guerraoui, R., Felber, P., Garbinato, B., and Mazouni, K., System Support for Objects Groups in *Proceedings of ACM Conference on Object Oriented Programming Systems*, Languages and Applications (OOPSLA'98) , October 1998 .

[47] Harchol, Mor, and Allen Downey, Exploiting Process Lifetime Distributions for Dynamic Load Balancing, in *ACM Transactions on Computer Systems*, volume 15, number 3, pages 253-285, August 1997.

[48] Heath, D., Resnick, S., and Samorodnitsky, G., Patterns of Buffer Overflow in a Class of Queues with Long Memory in the Input Stream, School of OR & IE technical report 1169, Cornell University, 1996.

[49] Horst, R.W., and Chou, T.C.K., An Architecture for High-Volume Transaction Processing, in *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, MA, 1995.

[50] Jacobson, V., Braden, R., and Borman, D., TCP Extensions for High Performance, IETF Request for Comments 1323, May 1992.

[51] Jurczyk, M., Performance and implementation aspects of higher order head-of-line blocking switch boxes, in *Proceedings of the 1997 International Conference on Parallel Processing*, 1997., 1997 , Page(s): 49 -53.

[52] Katevenis, M., Serpanos, D., Spyridakis, E., Switching Fabrics with Internal Backpressure using the Atlas I Single Chip ATM Switch, in *Proceedings of the GLOBECOM'97 Conference*, Phoenix, AZ, Nov. 1997.

[53] Katzman, J.A., et al, A Fault-tolerant multiprocessor system, United States Patent 4,817,091, March 28, 1989.

[54] Kay, J., and Pasquale, J., The importance of Non-Data Touching Processing Overheads, in *Proceedings of ACM SIGCOMM-93*, pages 259-269, Aug. 1993.

[55] Kent, C., and Mogul, J., Fragmentation Considered Harmful, in *Proceedings of ACM SIGCOMM-87*, pages 390-410. Aug 1987.

[56] Kistler, James J., and M. Satyanarayanan, Disconnected Operation in the Coda File System, *ACM Transactions on Computer Systems*, volume 10, number 1, pages 3-25, February 1992. .

[57] Krishnamurthy, A., Lumetta, S., Culler, D. E. and Yelick, K., Connected Components on Distributed Memory Machines, DIMACS Series in *Discrete Mathematics and Theoretical Computer Science*.

[58] Kronenberg, N., Levy, H., and Strecker, W.,VAXclusters: A Closely Coupled Distributed System, *ACM Transactions on Computer Systems*, vol. 4, no. 2 (May 1986).

[59] Kuenning, G. H., Popek, G.J. and Reiher, P.L., An Analysis of Trace Data for Predictive File Caching in Computing, in *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 291-303, Boston, MA, June 1994.

[60] Landis, S.,and Maffeis, S. Building reliable distributed systems with CORBA, *Theory and Practice of Object Systems*, John Wiley & Sons, New York, 1997.

[61] Laubach, M., Classical IP and ARP over ATM, IETF Request for Comments 1577, January 1994.

[62] Maeda, C. and Bershad, B.N., Protocol Service Decomposition for High-Performance Networking, in *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 244-255. Dec. 1993.

[63] Maffeis, S., Adding group communication and fault-tolerance to CORBA, in *Proceedings of the Usenix Conference on Object-Oriented Technologies*, June 1994.

[64] Majumdar, S., and Bunt, R.B., Measurement and Analysis of Locality Phases in File Referencing Behavior, in *Proceedings of the 1986 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 180-192, Raleigh, NC, May 1986.

[65] Moser, L., Melliar-Smith, M., D. A. Agarwal, D., Budhia, R., and Lingley-Papadopoulos, C., Totem A Fault-Tolerant Multicast Group Communication System, *Communications of the ACM*, April 1996.

[66] Mummert, L.B., and M. Satyanarayanan, Long Term Distributed File Reference Tracing: Implementation and Experience, *Software: Practice and Experience*, volume 26, number 6, pages 705-736, June 1996.

[67] Nagar, R., *Windows NT File System Internals*, O'Reilly & Associates, September 1997.

[68] Narasimhan, P., Moser, L.E., and Melliar-Smith, P.M., Exploiting the Internet Inter-ORB Protocol interface to provide CORBA with fault tolerance, in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, Portland, OR, June 1997.

[69] Nelson, M.N., Welch, B.B., and Ousterhout, J.K., Caching in the Sprite Network File System, *ACM Transactions on Computer Systems*, volume 6, number 1, pages 134-154, February 1988.

[70] Nick, J.M., Moore, B.B., Chung, J.-Y., and Bowen, N., S/390 cluster technology: Parallel Sysplex*, IBM Systems Journal*, Vol32, No. 2, 1997.

[71] Ousterhout, John K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer M., and Thompson, J.G., A Trace-Driven Analysis of the UNIX 4.2BSD File System, in *Proceeding of the Tenth ACM Symposium on Operating Systems Principles*, pages 198-121, Orcas Island, WA, October 1991.

[72] Ozkasap, O., Renesse, R. van, Birman, K.P., and Zhen Xiao, Z., Efficient Buffering in Reliable Multicast Protocols, in *Proceedings of the International Conference on Networked Group Communication (NGC'99)*. Pisa, Italy, November 1999.

[73] Pakin, S., Lauria, M., and Chien, A., High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet, in *Proceedings of ACM SuperComputing '95*, ACM Press, New York, 1995.

[74] Partridge, C., and Pink, C., A Faster UDP, *IEEE/ACM Transactions on Networking*, Vol. 1, No. 4, pages 429-440, Aug. 1993.

[75] Pfister, G.F. and V.A. Norton, Hot Spot Contention and Combining Multistage Interconnection Networks, *IEEE Transactions on Computers*, C-34(10). 1985.

[76] Pfister, G.F., *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, Prentice Hall, 1995.

[77] Pfister, G.F.,*In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing*, Prentice Hall, 1998.

[78] Ramakrishnan, K. K., P. Biswas, and R. Karedla, Analysis of File I/O Traces in Commercial Computing Environments, in *Proceedings of the 1992 ACM SIGMETRICS and Performance `92 International Conference on Measurement and Modeling of Computer Systems*, pages 78-90, Pacific Grove, CA, June 1992.

[79] Ranganathan, S., George, A., Todd, R., and Chidester, M., Gossip-Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters, *Cluster Computing*, Vol. 4, No. 3, July 2001.

[80] Renesse, R. van, Birman, K., Hayden, M., Vaysburd, A., and Karr, D., Building Adaptive Systems Using Ensemble, *Software--Practice and Experience*, August 1998.

[81] Renesse, R. van, Yaron Minsky, Y., and Hayden, M., A Gossip-Based Failure Detection Service, in *Proceedings of ACM/IFIP Middleware `98*, Lancaster, England, September 1998.

[82] Renesse, R. van, Scalable and Secure Resource Location, in *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2000.

[83] Renesse, R. van, and Birman, K., Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management and Data Mining, submitted to *ACM Transactions on Computer Systems*, November 2001.

[84] Resnick, S.I., Heavy Tail Modeling and Teletraffic Data, school of OR & IE technical report 1134, Cornell University, 1995.

[85] Robertson, A., Linux-HA Heartbeat System Design, in *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GE, October, 2000.

[86] Romanow, A. and Floyd. S., Dynamics of TCP traffic over ATM networks, in *Proceedings of ACM SIGCOMM-94*, pages 79-88, Aug. 94.

[87] Rosenblum, M., and Ousterhout, J.K., The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems*, 10(1), pages 26-52, February 1992.

[88] Saito, Y., Bershad, B.N., and Levy, H.M, Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service, in *Proceeding of the 16th ACM Symposium on Operating Systems Principles*, Charleston, December 1999.

[89] Satyanarayanan, M., A Study of File Sizes and Functional Lifetimes, in *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 96-108, Pacific Grove, CA, December 1981.

[90] Schauser, K. E. and Scheiman. C. J., Experience with Active Messages on the Meiko CS-2, in *Proceedings of the 9th International Parallel Processing Symposium (IPPS'95)*, Santa Barbara, CA, April 1995.

[91] Scott, S.L.; Sohi, G.S., The use of feedback in multiprocessors and its application to tree saturation control, *IEEE Transactions on Parallel and Distributed Systems*, Volume: 1 4, Oct. 1990 , Page(s): 385 -398.

[92] Seltzer, M., Krinsky, D., Smith, K., and Zhang, X., The Case for Application-Specific Benchmarking, in *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems*, Rico, AZ, 1999.

[93] Singhai, A., Sane, A., and Campbell, R.H., Quarterware for Middleware, *Proceedings of the International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, 1998.

[94] Smith, A.J., Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering*, volume 7, number 4, pages 403-417, July 1981. .

[95] Smith. B.C., Cyclic-UDP: A Priority-Driven Best-Effort Protocol, http://www.cs.cornell.edu/Info/Faculty/bsmith/nossdav.ps.

[96] Solomon, D., *Inside Windows NT*, Second Edition, Microsoft Press, 1998.

[97] Sterling, T.L., Salmon, J., Becker, D.J., Savarese, D.F., *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*, MIT press, June 1999.

[98] Thekkath, C. A., Levy, H. M., and Lazowska, E. D., Separating Data and Control Transfer in Distributed Operating Systems, in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1994.

[99] Vogels, W., World Wide Failures, in *Proceedings of the 7th ACM SIGOPS European Workshop*, Conamora, Ireland, September 1996.

[100] Vogels, W., Dumitriu, D., Birman, K. Gamache, R., Short, R., Vert, J., Massa, M., Barrera, J., and Gray, J., The Design and Architecture of the Microsoft Cluster Service -- A Practical Approach to High-Availability and Scalability, in *Proceedings of the 28th IEEE Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998.

[101] Vogels, W., Dumitriu, D., Agrawal, A., Chia, T., and Guo K., Scalability of the Microsoft Cluster Service, in *Proceedings of the Second Usenix Windows NT Symposium*, Seattle, WA, August 1998. .

[102] Vogels, W., van Renesse, R., and Birman, K., Six Misconceptions about Reliable Distributed Computing, in *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[103] Vogels, W., Chipalowsky, K., Dumitriu, D., Panitz, M., Pettis, J., Woodward, J., Quintet, tools for building reliable distributed components, *in Proceedings of the 2nd International Enterprise Distributed Object Computing Workshop*, San Diego, November, 1998.

[104] Wang, Y., and Lee, W., COMERA: COM Extensible Remoting Architecture, in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, Santa Fe, NM, April 1998.

[105] Welsh, M., Basu, A., Huang, X.W. and Eicken T. von, Memory Management for User-Level Network Interfaces, *IEEE Micro*, Mar.-Apr. 1998, pp. 77-82.

[106] Willinger, W., and Paxson, V., Where Mathematics meets the Internet, in *Notices of the Amercian Mathematical Society*, volume 45, number 8, 1998.

[107] Zhou, S., Da Costa, H., and Smith, A.J., A File System Tracing Package for Berkeley UNIX, in *Proceedings of the USENIX Summer 1985 Technical Conference*, pages 407-419, Portland Oregon, June 1985.

Samenvatting

In de afgelopen tien jaar heeft Enterprise Computing een aantal belangrijke veranderingen doorgemaakt. Voorheen lag de nadruk op een centrale verwerking van gegevens door een beperkt aantal database servers, voornamelijk ter ondersteuning van online transaction processing en verwerking van batch-opdrachten. Deze gegevensverwerking geschiedt nu door groepen applicatie-servers die een grote verscheidenheid van diensten verzorgen voor de verschillende bedrijfsprocessen. Inbegrepen in deze diensten zijn nog steeds de traditionele orderverwerking en inventarisbeheer, maar zij bevatten nu ook de interne en externe informatie portals, verschillende email- en samenwerkingsdiensten, de alom doorgedrongen integratie van customer relationship management in allerlei processen, alsmede diverse rekendiensten ter ondersteuning van de enterprise management processen zoals bijvoorbeeld de financiële voorspellings-technologie en het ononderbroken verwerken van de gegevens van de onderneming door datamining-processen.

Het is van belang te onderkennen dat deze diensten centraal zijn komen te staan in het functioneren van de onderneming en dat vertraging van of het volledig uitvallen van deze diensten de hele onderneming kan stil leggen. Deze diensten zijn dus van mission-critical aard voor het functioneren van de onderneming en als zodanig dienen zij schaalbaar en foutbestending te zijn en hun prestaties zo volledig mogelijk gegarandeerd. Om aan deze eisen tegemoet te kunnen komen, leek het de aangewezen weg om deze diensten te organiseren in compute-clusters, aangezien deze technologie de mogelijkheid bood om zowel schaalbaar als foutbestendig te opereren.

In het begin van de jaren negentig was de cluster-computing technologie voornamelijk beperkt tot OLTP en parallel computing. Deze technologie was onvoldoende om de schaalbare en betrouwbare diensten te ontwikkelen die de nieuwe informatie georiënteerde onderneming nodig had. De problemen waarmee enterprise cluster computing geconfronteerd werden zijn het best weergegeven door Greg Pfister in zijn boek "In Search of Clusters ...":

> “*Pogingen om parallel processing te gebruiken zijn in het verleden gestruikeld over slappe micro-processoren, overbodige communicatie-patronen en de noodzaak om de parallelle programmatuur iedere keer weer opnieuw van de grond af op te bouwen. Deze situatie heeft geleid tot de volledig begrijpelijke overtuiging dat deze vorm van gegevensverwerking geen toekomst had indien niet enorme vooruitgang zou worden geboekt in prestaties of functionaliteit*”

In dit proefschrift worden de resultaten beschreven van mijn onderzoek naar oplossingen voor een aantal van de problemen waarmee mission-critical enterprise cluster-computing geconfronteerd werd. Deze problemen leken aanvankelijk onoverkomelijke barrières voor de wijdverspreide introductie van cluster-computing in de enterprise. Mijn onderzoek heeft zich voornamelijk toegespitst op de volgende vier gebieden:

- De eliminatie van de barrières die het gebruik van hoge-snelheids netwerken in standaard werkstations en servers in de weg stonden.
- Het ontwerp van efficiënte runtime-systemen voor cluster-bewuste enterprise-applicaties.
- Het opzetten van een gestructureerd beheer van grootschalige enterprise cluster-computing systemen.
- De analyse van grootschalige systemen door middel van instrumentatie en gebruiks-monitoring.

De resultaten van mijn onderzoek, als beschreven in dit proefschrift hebben in grote mate bijgedragen aan het creëren van de mogelijkheid om nu wel cluster-applicaties te bouwen die in staat zijn om de diensten te verlenen die tegemoet komen aan de schaalbaarheids- en beschikbaarheids eisen van de moderne onderneming. De ontworpen technologieën hebben hun toepassing gevonden in industrie standaards zoals de Virtual Interface Architecture, zijn gebruikt in commercieel beschikbare applicatie-servers en zijn gebruikt voor het ontwerp van een nieuw commercieel, grootschalig cluster-beheerssysteem, dat ondersteuning geeft aan enterprise-wide, geografisch verspreid beheer van cluster diensten. In dit proefschrift wordt een overzicht gegeven van de meest belangrijke onderzoeksresultaten. Echter het aantal resultaten waartoe dit onderzoek heeft geleid is veel groter, en strekt zich uit ook buiten deze vier gebieden. Deze zijn terug te vinden middels de referenties in Appendix A.

Een aantal onderdelen van de Quintet software zijn overgenomen door commerciële applicatie servers, en zij vormen ook de basis voor de applicatie ondersteuning in het Galaxy cluster-beheerssysteem. Sommige specifieke onderdelen zoals de multi-level failure detector zijn opnieuw gebruikt in cluster-beheerssystemen, maar ze zijn ook beschikbaar gemaakt als op zichzelf staande software voor het onderhouden van kleinere web- en compute-clusters.

In deel II van dit proefschrift wordt mijn onderzoek naar cluster-runtime en cluster-beheerssystemen behandeld. In hoofdstuk 5 worden de ervaringen met het toepassen van academisch ontwikkelde software in productie systemen beschreven, terwijl in hoofdstuk 6 een overzicht wordt gegeven van de Quintet applicatie server.

Gestructureerd beheer van grootschalige enterprise cluster- computing systemen

De verbeteringen in zowel de processor- als de communicatietechnologieën in het begin van de jaren negentig waren op zichzelf niet voldoende om cluster-computing beschikbaar te maken voor de algemene enterprise-computing wereld. Software ondersteuning voor enterprise cluster-computing, zowel op het systeem- als op het applicatie niveau, werd nog steeds ernstig geplaagd door structurerings- en schaalbaarheidsproblemen. Dit was het duidelijkst op het terrein van de cluster-beheerssystemen, waar de traditionele beheerssystemen waren ontworpen voor zeer kleinschalige systemen, en vaak gebaseerd op een zeer specifiek hardware platform.

Waar de kosten-effectieve parallelle gegevensverwerking op grote schaal werd mogelijk gemaakt door het Beowulf cluster-beheerssysteem, was er geen vergelijkbare oplossing voor mission-critical enterprise computing. De voornaamste reden voor het ontbreken van enige vooruitgang op dit gebied was het feit dat een cluster-beheerssysteem voor enterprise-computing een grote variëteit van applicatietypes moest ondersteunen, ieder met zeer specifieke eisen met betrekking tot de uitvoering van schaalbaarheid en beschikbaarheid. De software-technologie die nodig was om deze ondersteuning te geven was in vele gevallen complexer dan de applicatie-software waarvan het de ondersteuning diende te zijn.

Het Galaxy Cluster Management Framework was het eerste beheerssysteem dat een schaalbare oplossing bood voor het beheer van clusters in grootschalige datacenters. In Galaxy is de beheers- infrastructuur opgebouwd uit verschillende lagen, waardoor het mogelijk is om grote groepen (farms) van clusters te beheren terwijl deze farms

Hieronder volgt nu een meer gedetailleerde beschrijving van elk van de vier onderzoeksgebieden.

Barrières bij het gebruik van hoge-snelheids netwerken door standaard werkstations

Hoewel er in het begin van de jaren negentig goede vooruitgang was geboekt in de ontwikkeling van hoge-snelheids cluster interconnects, was de technologie nog niet gereed om gebruikt te gaan worden door kant-en-klare enterprise cluster-systemen. De communicatie technieken waren volledig gericht op de specifieke wijze waarop besturings-systemen hun diensten aanboden aan de applicaties en de wijze waarop parallelle applicaties ontworpen werden. De traditionele technieken maakten het onmogelijk om de interconnect technologie te gaan gebruiken in standaard werkstations. Een voorbeeld hiervan is de IBM SP2, die alleen maar aan één applicatie toestond om de interconnect te gebruiken, aangezien het besturingssysteem geen bescherming en isolatie bood voor deze vorm van netwerk communicatie.

Met de komst van een aantal gestandaardiseerde hoge-snelheids netwerk technologieën werd de verwachting gewekt dat het op korte termijn mogelijk zou worden voor reguliere werkstations en servers om gelijkwaardige communicatie technologieën te gebruiken als de parallelle verwerkingssystemen, maar dan wel op een veel kosten-effectievere manier. Helaas waren de besturingssystemen van deze computers niet ontworpen voor hoge-snelheids communicatie en de vertragingen bij de verwerking van netwerk-pakketten door het besturingssysteem waren dusdanig groot dat de meeste voordelen van de nieuwe netwerken niet beschikbaar konden worden gemaakt voor de applicaties.

Tezamen met Thorsten von Eicken was ik in 1994 begonnen aan een onderzoek gericht op het doorbreken van deze barrières. Dit onderzoek resulteerde in een nieuwe communicatiestructuur voor besturingssystemen, genaamd U-Net, dat een volledig nieuwe abstractie bood en de kracht van de user-level communicatie combineerde met de volledige bescherming en isolatie van traditionele besturingssystemen. In U-Net was de netwerk adapter gevirtualizeerd in de adresruimte van de applicatie, waardoor het mogelijk werd om prestaties te bereiken die dicht bij de maximaal haalbare hardware-prestaties lagen. Doordat in deze structuur het data transport en de controle mechanismen voor de netwerk processen volledig gescheiden waren was het mogelijk om cluster applicaties te ontwikkelen met zeer goede prestaties.

Een industrie consortium onder leiding van Intel, Microsoft en Compaq nam het U-Net prototype en gebruikte het als basis voor de Virtual Interface Architecture, hetgeen nu de de-facto standaard is voor enterprise cluster-interconnects. De U-Net architectuur, de overgang van prototype naar een industrie-standaard en de ervaringen met een grootschalig cluster gebaseerd op de VIA technologie, zijn beschreven in deel I van het proefschrift.

Efficiënte runtime systemen voor cluster-bewuste enterprise applicaties

Door de voortuitgang die geboekt was in de verbetering van de schaalbaarheid van cluster-hardware en besturingssystemen konden een groter aantal applicaties nu gebruik maken van de beschikbare netwerk snelheden en van de grotere beschikbaarheid van de systemen. De structuur van de meeste van deze applicaties was echter gebaseerd op de afwezigheid van een gedistribueerde systeemstructuur en er was geen mogelijkheid om nieuwe communicatie-abstracties te integreren. Hoewel dit transparant houden van de distributie een aantal voordelen leek te hebben, voornamelijk in de interactie tussen clients en servers, was het introduceren van nieuwe technologieën onvermijdelijk om tegemoet te kunnen komen aan de eisen van schaalbaarheid en beschikbaarheid op het niveau van de server applicaties. Deze conclusie was mede gebaseerd op de jarenlange ervaring met het ontwerpen en toepassen van groep-communicatie systemen voor complexe, gedistribueerde productie systemen, waar het duidelijk was dat het volledig afgeschermd houden van de distributie eigenschappen een grote hindernis was voor het construeren van gevorderde server applicaties.

Mijn onderzoek naar efficiënte runtime systemen voor cluster-bewuste applicaties concentreerde zich op de vraag wat de juiste software-hulpmiddelen zouden moeten zijn die de applicatie-ontwikkelaars ten dienste zouden moeten staan, indien het gedistribueerde karakter van de applicatie expliciet zou zijn in plaats van afgeschermd. Het resultaat van dit onderzoek was een voor cluster applicaties gespecialiseerde applicatie server, genaamd “Quintet”. Deze Quintet applicatie-server geeft de ontwikkelaar van enterprise componenten gereedschap in handen om deze componenten zowel schaalbaar als extra beschikbaar te laten zijn. Deze extra diensten zijn echter niet afgeschermd en de ontwikkelaar dient een aantal mechanismen aan de componenten toe te voegen ter assistentie bij het uitvoeren van de gedistribueerde taken door het runtime systeem.

mogelijk over verschillende geografische lokaties zijn verdeeld. Binnen iedere farm bevinden zich kleinere groepen van gespecialiseerde clusters die ieder beheerd worden naargelang hun cluster profiel. Het systeem was ontworpen overeenkomstig de principes die voortgekomen waren uit de analyse van het grootschalig gebruik van groep-communicatie systemen, alsmede uit een diepgaande analyse van bestaande enterprise cluster-beheerssystemen.

Galaxy kon worden beschouwd als een succes toen het ontwerp en onderliggende principes door één van de grootste besturingssysteem-bedrijven werd overgenomen als basis voor hun next-generation cluster beheerssysteem.

Het voorbereidende werk en een overzicht van het Galaxy cluster-beheerssysteem is terug te vinden in hoofdstukken 7 en 8 van dit proefschrift.

Analyse van grootschalige systemen.

Het verkrijgen van inzicht in de wijze waarop software systemen worden gebruikt in de dagelijkse praktijk is essentieel voor systeem georiënteerd onderzoek dat tot doel heeft bij te dragen aan het vinden van oplossingen voor de problemen waarmee de huidige informatie onderneming geconfronteerd wordt. Zowel het instrumenteren van systemen en verzamelen van gebruiksgegevens op grote schaal over langere tijd, als het analyseren van deze gegevens is een onderzoeksgebied op zich. Het ontwerpen van het monitor- en analyse systeem is vaak zeer complex aangezien er vele gegevensbronnen zijn die gelijktijdig actief zijn, terwijl er geen controle is over het systeemgebruik dat de aanmaak van de gegevens veroorzaakt. Het is dan ook van het allergrootste belang om mechanismen te ontwerpen die dusdanig gegevens verzamelen dat er een precieze statistische analyse valt te maken, zonder dat dit proces invloed uitoefent op de werkwijze van het systeem dat bestudeerd wordt.

In dit proefschrift zijn een tweetal studies opgenomen die op grote schaal het gebruik van systemen onderzoeken:

De eerste studie is een rapportage van de resultaten van een onderzoek naar het gedrag van een hogesnelheid cluster-interconnect wanneer deze wordt overbelast. De cluster-interconnect is opgebouwd uit 40 op VIA gebaseerde netwerk-switches. De studie maakt gebruik van een traditioneel experimentmodel, waarin er volledige controle is op de wijze waarop de netwerk-last gegenereerd wordt. Het unieke aan dit experimentmodel was echter dat het dusdanig ontworpen moest worden dat van de observaties aan de eindpunten afgeleid kon worden wat het gedrag van de switches in het midden van het netwerk was, aangezien het niet mogelijk was om de switches zelf te instrumenteren. Het ontwerp van het experimenteel raamwerk vereiste speciale aandacht aangezien er met hoge snelheid zeer grote hoeveelheden metingen dienden te worden verricht. De details van deze studie zijn terug te vinden in hoofdstuk 4.

In de tweede studie, waaraan deel 3 van dit proefschrift gewijd is, wordt de aandacht gericht op het probleem dat optreedt wanneer er geen controle over de bronnen mogelijk is. Het onderwerp van de studie was een grootschalig onderzoek naar de gebruiks-karakteristieken van het file-systeem van reguliere werkstations. Hiertoe was een groot aantal werkstations geïnstrumenteerd en geobserveerd gedurende een langere periode. De analyse van de resultaten werd ernstig bemoeilijkt door de enorme hoeveelheid observaties en de complexiteit die geïntroduceerd werd door de heterogeniteit in de applicaties die actief waren op de verschillende werkstations. In de conclusies van deze studie wordt de nadruk gelegd op het gebruik van speciale statistische technieken met betrekking tot de analyse van grote verzamelingen observaties. Deze exacte statistische benadering, die ontbrak in vroegere file-systeem-studies, is noodzakelijk voor het ontwerp van de juiste werklast-modellen, die gebruikt kunnen worden in toekomstige file-systeem prestatietesten en bij het ontwerpen van nieuwe file-systeem-componenten.

Appendix A

Publications

2004

Technology Challenges for the Global Real-Time Enterprise
Werner Vogels, in the Journal of Knowledge Management, Volume 8, Issue 1, January 2004

2003

Web Services are not Distributed Objects: Common Misconceptions about the Fundamentals of Web Service Technology
Werner Vogels, IEEE Internet Computing, Vol. 7, No. 6, November/December 2003.

Tracking Service Availability in Long Running Business Activities
Werner Vogels, in Proceeding of the First International Conference on Service Oriented Computing (ICSOC 2003), Trento, Italy, December 2003

Benchmarking CLI-based Virtual Machines
Werner Vogels, in IEE Proceedings - Software, Volume 150, Issue 6, October 2003

HPC.NET - are CLI-based Virtual Machines Suitable for High-Performance Computing?
Werner Vogels, in Proceeedings of the 15th Supercomputing Conference (SC2003), Phoenix, AZ, November 2003

Scalability and Robustness in the Global Real-Time Enterprise
Werner Vogels, in Proceedings of the 2003 Workshop on High-Performance Transaction Processing (HPTS 2003), Asilomar, CA, October 2003

Navigating in the Storm: Using Astrolabe for Distributed Self-Configuration, Monitoring and Adaptation
Ken Birman, Robbert van Renesse, and Werner Vogels, in Proceedings of the Fifth Annual International Workshop on Active Middleware Services (AMS 2003), Seattle, WA, June 2003.

WS-Membership - Failure Management in a Web-Services World
Werner Vogels and Chris Re, in the Proceedings of the Twelfth International World Wide Web Conference, Budapest Hungary, May 2003.

Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management and Data Mining
Robbert van Renesse, Ken Birman and Werner Vogels, in the ACM Transaction on Computer Systems, Volume 21, Issue 2, pages 164-206, May 2003

2002

The Power of Epidemics: Robust Communication for Large_sclae Distributed Systems
Werner Vogels, Robbert van Renesse, and Ken Birman, in Proceedings of the ACM First Workshop on Hot Topics in Networks (HotNets-I), Princeton, NJ, October 2002.

Scalable Data Fusion Using Astrolable
Ken Birman, Robbert van Renesse and Werner Vogels, in Proceedings of the Fifth International Conference onInformation Fusion 2002 (IF 2002), Annapolis, MD, July 2002.

A Collaborative Infrastructure for Scalable and Robust News Delivery
Werner Vogels, Chris Re, Robbert van Renesse, and Ken Birman, in Proceedings of the IEEE Workshop on Resource Sharing in Massively Distributed Systems (RESH'02), Vienna, Austria, July 2002.

Technology Challenges for the Global Real-Time Enterprise
Werner Vogels, in Proceedings of the International Workshop on Future Directions in Distributed Computing, Bertinoro, Italy, June 2002.

Scalable Management and Data Mining Using Astrolabe
Robbert van Renesse, Ken Birman, Dan Dumitriu and Werner Vogels, in

Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, MA, March 2002.

2001

Spinglass, Scalable and Secure Communication Tools for Mission-Critical Computing
Ken Birman, Robbert van Renesse, Werner Vogels, in Proceeedings of the 2001 DARPA Information Survivability Conference and Exhibition - II, June 2001.

Using Epidemic Techniques for Building Ultra-Scalable Reliable Communication Systems
Werner Vogels, Robbert van Renesse, and Ken Birman, in Proceedings of the Large Scale Networking Workshop: Research and Practice, Vienna, VA, March 2001.

2000

An Overview of the Galaxy Management Framework for Scalable Enterprise Cluster Computing
Werner Vogels and Dan Dumitriu, in the Proceedings of the IEEE International Conference on Cluster Computing: Cluster-2000, Chemnitz, Germany, December 2000.

Tree Saturation Control in the AC3 Cluster interconnection
Werner Vogels, David Follet, Jen Wei, David Lifka and David Stern, in Proceedings of the 8th IEEE Hot Interconnets Symposium, Stanford, CA, August 2000.

The Horus and Ensemble Projects: Accomplishments and Limitations
Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christopher Kreitz, Werner Vogels, Robbert van Renesse and Ohad Rodeh, in Proceedings of the 2000 DARPA Information Survivability Conference and Exhibition, Hilton Head, SC, January 2000.

1999

File System Usage in Windows NT 4.0
Werner Vogels, in Proceedings of the 17th ACM Symposium on Operating Systems Principles, Kiawah Island, SC, December 1999.

Windows 2000 Research Edition, Where the Academic White Knights meet the Evil Empire...
Werner Vogels, Informatik/Informatique, Swiss Computer Society, February 1999.

1998

Quintet, Tools for Reliable Enterprise Computing,
Werner Vogels, Dan Dumitriu, Mike Panitz, Kevin Chipalowsky and Jason Pettis, in Proceedings of the 2nd International Enterprise Distributed Object Computing Conference, San Diego, November, 1998.

Evolution of the Virtual Interface Architecture,
Thorsten von Eicken and Werner Vogels, IEEE Computer, Volume 35 Number 11, November 1998.

Six Misconceptions about Reliable Distributed Computing
Werner Vogels, Robbert van Renesse and Ken Birman, in Proceedings of the 8th ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.

Scalability of the Microsoft Cluster Service
Werner Vogels, Dan Dumitriu, Anand Agrawal, T. Chia, and Katerine Guo, in Proceedings of the Second Usenix Windows NT Symposium, Seattle, WA, August 1998.

The Design and Architecture of the Microsoft Cluster Service -- A Practical Approach to High-Availability and Scalability
Werner Vogels, Dan Dumitriu, Kenneth Birman, Rod Gamache, Rob Short, John Vert, Mike Massa, Joe Barrera, and Jim Gray, in Proceedings of the 28th symposium on Fault-Tolerant Computing, Munich, Germany, June 1998.

Building Reliable Distributed Components -- or the case for anti-transparent reliability tools
Werner Vogels, Dan Dumitriu and Mike Panitz, Usenix COOTS 1998 Advanced Workshop on Scalable Object Systems, May 1998.

1997

GSGC: An Efficient Gossip-Style Garbage Collection Scheme for Scalable Reliable Multicast
Katherine Guo, Mark Hayden, Werner Vogels, Robbert van Renesse, and Ken Birman, Technical Report Department of Computer Science Cornell University CS-TR-97-1656, December 1997.

Object Oriented GroupWare using the Ensemble System
Werner Vogels, in Proceedings of the 5th European Conference on Computer Supported Collaborative Work, Workshop on Object Oriented Groupware Platforms, Lancaster, UK, September 1997.

Hierarchical Message Stability Tracking Protocols
Katherine Guo, Werner Vogels, Robbert van Renesse, and Ken Birman, Technical Report Department of Computer Science Cornell University, CS-TR-971647, September 1997.

Moving the Ensemble Groupware System to Windows NT and Wolfpack
Ken Birman, Werner Vogels, MarkHayden, Katherine. Guo, Takako. Hickey, Roy. Friedman, Silvano. Maffeis, Robbert. van Renesse, and Alexey. Vaysburd, in Proceedings of the 1997 USENIX Windows NT Workshop, Seattle, Washington, August 1997.

1996

World-Wide Failures
Werner Vogels, in Proceedings of the 1996 ACM SIGOPS Workshop, Connemora, Ireland, September 1996.

Structured Virtual Synchrony: Exploring the bounds of Virtual Synchronous Group Communication
Katherine Guo, Robbert van Renesse and Werner Vogels, in Proceedings of the 1996 ACM SIGOPS Workshop, Connemora, September 1996.

1995

U-Net: A User-Level Network Interface for Parallel and Distributed Computing
Thorsten von Eicken, Anindya Basu, Vineet Buch and Werner Vogels in Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, December 1995,

Delivering High-Performance Communication to the Application-Level
Werner Vogels and Thorsten von Eicken, in Proceedings of the Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95), August 1995.

Horus: A Flexible Group Communications System
Robbert van Renesse, Kenneth P. Birman, Brad Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd and Werner Vogels, Technical Report Department of Computer Science Cornell University, CS-TR 95-1500, March 23, 1995

1993

The Changing Face of Technology in Distributed Systems
Paulo Verissimo and Werner Vogels, in Proceedings of the 4th Workshop on Future Trends in Distributed Computing, Lisboa, Portugal, 1993.

Reliable Group Communication in Internetworks using LAN-oriented Protocols
Werner Vogels and Paulo Verissimo, in Proceedings of the Fifth European Workshop on Dependable Computing, Lisboa, Portugal, 1993.

Group Orientation: a Paradigm for Modern Distributed Systems
Paulo Verissimo, Luis Rodrigues and Werner Vogels, in First Year Report of the Broadcast ESPRIT/BRA project, Newcastle upon Tyne, United Kingdom, 1993.

A Framework for Structuring group support in LSDC's,
Werner Vogels, Luis Rodrigues and Paulo Verissimo, in First Year Report of the Broadcast ESPRIT/BRA project, Newcastle upon Tyne, United Kingdom, 1993.

1992

Requirements for High-Performance group Support in Distributed Systems
Werner Vogels, Luis Rodrigues and Paulo Verissimo, in Proceedings of the 1992 ACM SIGOPS European workshop, Mont Saint Michel, France, 1992.

Fast Group Communication for Standard Workstations
Werner Vogels, Luis Rodrigues and Paulo Verissimo, in Proceedings of the OpenForum'92 Technical Conference, Utrecht, The Netherlands, 1992.

1991

Supporting process groups in Internetworks with lightweight reliable multicast protocols
Werner Vogels and Paulo Verissimo, in Proceedings of the ERCIM Workshop on Distributed System, Lisboa, Portugal, 1991.

Appendix B

Additional Acknowledgements and Support

The publications that make up the chapters of this thesis often acknowledged the help of individuals and funding organisations.

Chapter 2 - U-Net: A User-Level Network Interface for Parallel and Distributed Computing.

U-Net would not have materialized without the numerous discussions, the many email exchanges, and the taste of competition we had with friends in the UC Berkeley NoW project, in particular David Culler, Alan Mainwaring, Rich Martin, and Lok Tin Liu.

The Split-C section was only possible thanks to the generous help of Klaus Eric Schauser at UC Santa Barbara who shared his Split-C programs and provided quick access to the Meiko CS-2 which is funded under NSF Infrastructure Grant CDA-9216202. The CM-5 results were obtained on the UCB machine, funded under NSF Infrastructure Grant CDA-8722788. Thanks also to the UCB Split-C group for the benchmarks, in particular Arvind Krishnamurthy.

Most of the ATM workstation cluster was purchased under contract F30602-94-C-0224 from Rome Laboratory, Air Force Material Command. Werner Vogels is supported under ONR contract N00014-92-J-1866. We also thank Fore Systems for making the source of the SBA-200 firmware available to us.

In the final phase several reviewers and Deborah Estrin, our SOSP shepherd, provided helpful comments, in particular on TCP/IP.

Chapter 3 - Evolution of the Virtual Interface Architecture

This research is funded by the US Defense Advanced Research Projects Agency's Information Technology Office contract ONR-N00014-92-J-1866, NSF contract CDA-9024600, a Sloan Foundation fellowship, and Intel Corp. hardware donations.

Chapter 4 - Tree Saturation Control in the AC3 Velocity Cluster Interconnect

This work was performed by the AC3x study group with participation from Cornell University (Theory Center and Computer Science), GigaNet Inc., Dell Corporation and the Server Architecture Lab of Intel Corporation. The Network Analyses were performance on the Velocity Cluster, a high-performance computing resource of the Cornell Theory Center.

The study group is very grateful for the help of George Coulouris and Resa Alvord during the execution of the experiments. A special thanks goes to Shawn Clayton of GigaNet Inc., for the assistance in designing the experiments.

Werner Vogels is supported by the National Science Foundation under Grant No. EIA 97-03470, by DARPA/ONR under contract N0014-96-1-10014 and by grants from Microsoft Corporation.

Chapter 6 - Scalability of the Micosoft Cluster Service

Discussions with Jim Gray, Catharine van Ingen, Rod Gamache and Mike Massa have helped to shape the research reported in this paper. The advice of shepherd Ed Lazowska was very much appreciated. Thorsten von Eicken, S. Keshav and Brian Smith graciously contributed hardware to the world's largest wolfpack cluster.

Chapter 8 - The Galaxy Framework for the Scalable Managament of Enterprise-Critical Cluster Computing

First and foremost our thanks go to Jim Gray for his continuous support over the years of this research. Without his advice, insights and intellectual stimulation the Galaxy architecture would never have risen to meet real world needs.

The large group of students who have implemented parts of Galaxy over the years deserve special credit, especially Dan Dumitriu who was responsible for the

implementation of a significant part of the distributed systems technology used in Galaxy.

Chapter 9 - File System Usage in Windows NT 4.0

Thanks goes to Fred Schneider for the motivation to do this study, to Thorsten von Eicken for the initial discussions of the results, and to my colleagues Ken Birman and Robbert van Renesse for never questioning what I was doing “on-the-side”.

The paper improved significantly because of the very thoughtful comments of the reviewers and the help of Peter Chen, the official shepherd. Special praise goes to Sara Shaffzin, without whose editorial help this paper would never have made it through the first round. Tim Clark, Raoul Bhoedjang and Ken Birman provided useful help with the final version.

This investigation was made possible by grants from Microsoft Corporation and Intel Corporation, and was supported by the National Science Foundation under Grant No. EIA 97-03470 and by DARPA/ONR under contract N0014-96-1-10014.