Pegasus — Operating System Support for Distributed Multimedia Systems

Ian M. Leslie¹

Derek McAuley¹

Sape J. Mullender²

November 16, 1992

1 Introduction

Pegasus³ is a project of the Universities of Cambridge (GB) and Twente (NL). This is a preliminary paper whose primary function is to state the goals of the project and to describe how we plan to set about reaching them.

The theme of the Pegasus project is the design and implementation of general-purpose operating-system support for distributed multimedia applications. It is our contention that for a system to be a multimedia system, it should endow text, images, audio and video with equal status: Interpreting an audio or video stream should not be a privileged task of special functions provided by the operating system, but one of ordinary user programs. If programs with no special status can process text interactively, then processing audio or video interactively should also be done by programs with no special status.

Very few, if any systems exist today that can claim to be multimedia systems in this sense. There is a wide range of multimedia applications on personal computers, but they are usually made to work by “taking over” the machine; that is, by not allowing other processes to run simultaneously, and possibly by directly addressing the hardware. Other systems offer only a very restricted set of multimedia applications, such as showing video in a window, or editing audio files. On these systems, as a rule, the video is not made available to user processes and the audio cannot be processed interactively. Often the video is analogue and often the network does not have the bandwidth nor the response time to transmit digital audio and video.

Our goal is the design and implementation of an operating system that allows capture, rendering, storage, and interactive processing of multimedia data by user-level applications, while keeping all of the desirable properties of distributed systems, such as resource sharing, data sharing, security, and fault tolerance.

These goals are ambitious, but we are greatly helped by technology, fast RISC processors, cheap large memories, hardware image and video compression, and gigabit networks. We intend to use high-performance technology, because we expect that today’s high-end technology will be personal-computer technology in ten years’ time.

¹. Cambridge University Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, United Kingdom

². University of Twente, Faculty of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands

³. Pegasus is partially supported by ESPRIT (BRA 6586), Olivetti Research Laboratory, Cambridge, Xerox Europarc, Cambridge, and Hewlett-Packard Laboratories, Palo Alto.

Figure 1. The Pegasus Systems architecture will be comprised of multimedia workstations (based on a DAN), networked devices, such as cameras, storage servers, compute servers, specialized multimedia data processing servers, and a high-speed interconnecting ATM network

2 System Configuration

The Pegasus system will be a distributed one. Distribution allows resource sharing as well as the exploitation of specialized hardware. The system will be composed of six key components, which can be identified in Fig 1.

Network

An ATM network will provide the communication infrastructure used by the components. The network will provide bandwidth, delay and jitter guarantees to streams of traffic. The guarantees provided by the ATM network are low level guarantees upon which the Pegasus operating system will make its guarantees to applications.

Initially this network will be based on the *Fairisle* network (Leslie and McAuley [1991]), an ATM LAN which has been constructed by the University of Cambridge.

Further work in incorporating the *Rattlesnake* fault tolerant switching fabric (Smit, Havinga and Smit [1992]) being developed at the University of Twente will be carried out.

Multimedia terminals

The Pegasus multimedia terminal is really a very ordinary workstation running the Pegasus operating system. It will be capable of displaying many video streams simultaneously in as

many windows. Each video stream will be coming into the workstation through an ATM host interface. Similarly, it will be capable of mixing incoming audio streams into an audio output device.

All the complexity of attaching cameras and other video and audio devices to terminals has been moved outside the terminal into the network. The ATM host interface uses per-connection descriptor tables to determine where in memory received data must go. The descriptor tables have enough flexibility in their design that incoming video data can be sent directly into the video RAM associated with a video window. It is the responsibility of the operating system to manipulate the descriptor tables as windows are moved, resized, obscured, etc.

Where necessary, video is clipped to fit its window at the source of the video stream, so that what you don't see is not unnecessarily sent over the network. As a consequence, the total video traffic over the network and the host interface will not exceed the traffic needed for one full-screen video window⁴ by much.

Networked devices

It is our plan to attach specialized devices for multimedia, such as cameras, video compression/decompression units, sound systems, data gloves, etc., to the network. We are already experimenting with an "ATM camera" and we expect that it will be straightforward to build other ATM devices as well.

To connect these device to the terminal, we make use of the Desk-Area Network (Hayter and McAuley [1991]). At the heart of the DAN is an ATM switch. The communication path across this single switch is considered reliable; no protocol is needed to deal with cell loss or broken circuits. The DAN is under complete control of the workstation operating system and failure of any one component is considered to be a failure of the system. This is not a big restriction, because a broken DAN would render a workstation incommunicado in any case. It does, however, simplify the protocol software enormously.

One port of the DAN switch is connected to the workstation via the ATM host interface. One or two other ports connect to the local ATM network. The remaining ports, typically between 5 and 12, are available for the connection of ATM devices.

Multimedia servers

The interactive processing of multimedia streams requires requires a strong emphasis on latency and bandwidth guarantees in the support infrastructure. Once one has achieved this goal for general purpose processors and across the network, one can have a bank of processors which can control or perform various digital signalling processing tasks.

The multimedia processor bank is essentially a collection of server machines that can run client multimedia programs. Some of these processors may be dedicated to carrying out particular functions for which they may have specialized hardware — e.g., video compression and decompression, image blending, sound filtering.

Unix servers

In the same fashion that Unix file system support is now provided by sending file-I/O operations to an NFS file system on another machine, we plan to provide Unix operating system support for multimedia applications by sending Unix system calls to specialized Unix servers. Programs without the low-latency requirements of multimedia can run directly on Unix platforms, but programs designed to process multimedia data streams interactively

⁴. Roughly 200 Mbps

will run on one of the multimedia servers and ship their Unix calls to one of the Unix servers.

Storage servers

Storage is provided by a set of storage servers. These will store typed file-like objects. The type information is essential because it allows the storage servers to determine what services to provide on a file-by-file basis.

A file containing a video sequence, for instance, must be written and read at a guaranteed rate; it must not be cached, because it is always read sequentially and is larger than the cache. The storage server will support replication for fault tolerance and for performance. These are different. Replication for fault tolerance can be done at fairly low levels, such as by disk mirroring. Replication for performance is done by caching and placing replicas where files are read and adjusting the number of replicas to the rate of change of the file.

3 Memory Management

Both networks and processors are becoming faster at a rapid pace. The speeds of memories, both primary and secondary, lag behind. As a result, traditional, bus-based, memory-centric computer architectures may become obsolete. In memory-centric architectures, memory is the most important shared resource: processors and devices communicate through memory buffers, processors communicate with each other through shared mutexes or semaphores in memory.

Making communication between gigabit networks and thousand-MIPS processors more effective requires the avoidance of main memory as an intermediary between processor and network. This applies equally for communication between processors and other high-speed devices. Hayter and McAuley [1991] proposed Cambridge University's DAN architecture to do exactly this.

In a DAN-based system, such as depicted in Figure 2, communication between all system components is via the network. Communication over the DAN between components within a system box, and communication over DAN and LAN or WAN between components in different boxes is only different in performance and possibly in reliability. In fact, much of the distinction between shared-memory multiprocessors, tightly coupled multiprocessors (e.g., connection machine) and loosely coupled systems (e.g., network of workstations) disappears.

The emergence of wide-address-space architectures is creating new ways of exploiting these new network architectures — processes on different machines can share data structures in physical memory much more easily by sharing address spaces as well. In shared memory, data can be shared; in a shared address space, data structures containing pointers can also be shared (Chase et al. [1992a]; Chase et al. [1992b]).

Sharing virtual address spaces may influence programming practice more profoundly than sharing physical memory. In a modern processor architecture, physical memory is, in fact, only shared indirectly — fast processors must have on-chip caches and data sharing only works with the assistance of a coherency protocol. Sometimes this protocol is implemented in hardware (e.g., snoopy caches), but on high-speed RISC architectures it now has to be more often realized in software.

We are investigating the use of shared address spaces for local groups of like machines. There does not seem to be any advantage in sharing address spaces among heterogeneous machine architectures: the data representations are different anyway. Nor does there appear to be any great advantage in sharing address spaces world wide: 64 bits would not be large enough for such

Figure 2. A Desk-Area Network connects CPUs, memories, devices and wider-area networks in an integrated manner.

sharing, but, more importantly, it becomes infeasible, given an address, to find the physical object with that address.

Pegasus uses a shared address space for local groups of mutually trusted machines that share the same data representation. A Pegasus address space is long lived; that is, the address space survives processes and processor crashes. The operating system manipulates memory objects called *segments*. A segment occupies a contiguous range of virtual addresses. A segment does not have to exist completely or partially in physical memory — that is, it may, for instance, be paged in and out.

Most segments are linked to external objects that we shall refer to as *files*. Files are managed by a — possibly distributed — *service*. A segment is created when such a linkage is first established in an address space. How this is done is illustrated in Figure 3.

A process P makes an $attach(O@S)$ request to the kernel to obtain a segment containing O 's data (arrow a). The kernel establishes a connection to server S through which it can request O 's data (arrow b or c). File servers, such as S_1 and S_2 , must implement a standard bi-directional interface which effectively allows the physical memory occupied by attached segments to be used as a cache for files. The kernel decides how much memory will be allocated to each service (possibly after negotiation). The service will decide what will be cached.

Many of the file services will be distributed — a set of storage servers plus a set of cache servers. Processes will normally attach objects at local cache servers: P will $attach(O_2@S_1)$, for example. The cache server S_1 then fetches the data from storage server S_2 . Storage servers will normally keep their data in an “address-space independent” representation and cache servers will unmarshal the data from the storage server into the local machine representation. As a result, the file-system cache will be in unmarshalled form, resulting in more efficient sharing.

Figure 3. Process P_1 makes an *attach($O@S$)* request. The kernel then creates a local segment and makes a connection to server S to get the data for object O . The kernel returns the segment's address to P_1 .

When a file is attached multiple times in the same address space, all segments will reside at the same virtual address even if the data is kept in different physical memories. When the last process detaches the last segment for a file in the address space, the address range is deallocated. An attempt is made, however, not to reuse addresses for as long as possible in order to detect stale references to a detached segment.

4 Storage Management

Object storage in Pegasus has been tailored to efficient management of persistent objects and multimedia data. The philosophy is similar to that of Multics, where main memory was viewed as a cache for disk storage and disk storage as a cache for archival storage. Files on disk map onto segments in memory and vice versa. A name server, usually integrated with the directory tree in a volume of files, manages information concerning file type, replication and consistency.

By associating relevant type information with the file system, it becomes possible for specialized methods in the file system itself to exploit the particular properties of particular types of data.

Video data, for example, will be read and written at a constant rate. It is useless to cache video data, because video fragments are usually larger than the cache anyway so that they merely serve to wipe out all other information from it and there is no point in being able to reproduce video data any faster than the viewing rate (unless the data is copied which is a rare event). Video data is usually modified by cutting and pasting. This can be straightforwardly realized by allowing disk blocks of video data to appear in multiple video files.

We are experimenting with the Log-Structured File System from Sprite (Rosenblum and Ousterhout [1991]) in Pegasus, initially because of its performance, but now for other reasons as well.

LFS writes both data and metadata to a log; the log is buffered and written to disk in large *chunks* (typically 256 kilobytes). We intend to exploit this in two ways. The first, which has already been explored by others, is to compress a chunk just before writing it to disk and decompressing it when reading it back. Burrows et al. [1992] found that they could save up to 50% disk space by doing this at a very small performance penalty. If the compression were done in hardware, the performance of a file system with compression would even be better than that of one without. With CPU speeds increasing more rapidly than disk speeds, even software compression is likely to become a performance advantage at some point.

Another advantage of the log is that the file system produces very large contiguous write operations. These can be striped over several disks for higher data rates. We are currently experimenting with algorithms that stripe each chunk over n disks plus k parity disks in such a way that each disk writes one n 'th of the data and that the data can be recovered from any n out of $n + k$ disks.

A third advantage of the log structure is that video data can be collected in separate chunks which are written to the log before the "regular" chunk containing the video file's metadata (*i-node*). In this way, video data is kept together and can always be read back in chunk-sized read operations.

We are currently studying something we've called *iterative logging*, a technique where a data from a log in non-volatile RAM streams into a log on a disk or RAID and from there to a log in WORM storage. Caching and backup will all be integrated in one simple mechanism. An extra

advantage is that the backup system is on-line and that historic data is available in the regular name space, much like Plan 9 (Presotto et al. [1991]).

5 Processing Multimedia

The performance of modern computing devices and networks allows the consideration of applications which integrate various real time media, such as video and audio, into a distributed computing system. Applications can be identified which need to manipulate the media data in real time; to provide such an environment presents new challenges to systems designers.

While general purpose processors continue to increase in power, human perception capabilities remain static (and expectation of quality only grows slowly). The prospect that digital signal processing tasks will be performed by general purpose processors is very real — we will be moving away from an environment of dedicated, special purpose hardware.

However, the management of shared resources must allow these applications to run correctly. Here, correctness includes a notion of time, but may not be concerned with the precise results generated by the application at all points in time. In other words, it is may be more important for an application to produce an approximate, or indeed no, result on time than a precise but late result.

The notion of quality of service as found in communication networks where probabilistic guarantees are placed on latency, loss, and bandwidth are closer to this model of correctness than is a traditional real time system. Our approach to processing continuous media, and by extension, multimedia, is based upon a quality of service model. We believe that this approach will allow us to build upon the guarantees provided by the ATM networks providing the lower level communication facilities.

Traditional real time systems are often static in nature. A system is configured with a known workload which is never changed. Here we are concerned with a dynamic multiprogrammed workload more akin to traditional multiprogrammed operating systems which use statistical multiplexing (paging and timeslicing) to increase resource utilization and total system throughput.

Overload in such a system is a possibility. One may have overload due to a long term overcommitment of resources. In a system processing continuous media this is an error. One should avoid making long term commitments that overload the system by denying the resource allocation to the application. The application is essentially asking the operating system to be party to a contract. As examples, such a contract might be concerned with the scheduling of the processor(s) or with a commitment to maintain parts of objects at particular levels in the cache hierarchy. Refusing the contract is is similar to a network refusing to complete a call. It can also be argued that this concept of contracts is similar to the situation in dynamic real time systems. To an extent this is true; as in real time systems the aim is not raw speed or the highest efficiency, but *predictability* and making resource *guarantees*. However, in the case of continuous media these guarantees may be probabilistic rather than absolute (and their may be the possibility of renegotiating with applications as load or offered load changes.) We also speculate that in processing multimedia, the conflicts over locks which plague the calculations for real time system scheduling will not be a dominant issue. Real conflicts on locks are likely to be over media synchronization and will be in the domain of the application rather than the operating system.

Unlike classical real time systems, overloads may also be short term. If resource allocation is based on probabilistic models of resource utilization then occasionally demand will exceed the available resources. In the case of silence suppressed voice for example, it is possible for all voice streams to be active simultaneously. There must be a policy for claiming the exceeded resource back from an application.

This reclamation might be based on knowledge of the the applications. For example, in the case of a video phone application, although regular computation is still necessary for the audio stream, it is easy to reduce the total demand made on the system by reducing video quality and frame rate. The reduction in video quality fits the Imprecise Computation model (Liu and al. [1991]); in an imprecise computation, a periodic task is split into mandatory subtask which the system guarantees to execute, and an optional subtask which is executed if resources permit and improves the *quality* of an approximate and acceptable result from the mandatory task.

On the longer time scales, those analogous to call acceptance, a process similar to quality of service negotiation make take place, but this need not be within the operating system itself. For example an application which fails to acquire resources may inform the user who may then reduce the size of an existing video window and then indicate that the application should try again. Alternatively the application may request (via a manager akin to a window manager) that another application's resource consumption should be reduced. Applications will thus be expected to handle events which claim resources back from them — the notion of imprecise computation may have several layers of optional subtask which might be performed.

6 References

M. Burrows, C. Jerian, B. W. Lampson and T. P. Mann [April 1992], *On-line Data Compression in a Log-structured File System*, DEC Systems Research Center, DEC SRC Technical Report 85, 130 Lytton Ave., Palo Alto, CA 94301.

J. S. Chase, H. M. Levy, M. Baker-Harvey and E. D. Lazowska [March 1992a], *How to Use a 64-bit Virtual Address Space*, University of Washington, Dept. of Computer Science & Engineering , Technical Report 92-03-02, Seattle WA.

J. S. Chase, H. M. Levy, E. D. Lazowska and M. Baker-Harvey [October 1992b], *Lighweight Shared Objects in a 64-bit Operating System*, Proceedings of OOPSLA'92 Conference on Object Oriented Programming Systems, Languages, and Applications, Also appears as Technical Report 92-03-09, Department of Computer Science & Engineering, University of Washington, March 1992.

M. Hayter and D. McAuley [October 1991], *The Desk-Area Network*, ACM Operating System Review 25, 14—21.

I. M. Leslie and D. R. McAuley [September 1991], *Fairisle: An ATM Network for the Local Area*, Proceedings of SIGCOMM'91.

J. Liu and et al. [May 1991], *Algorithms for Scheduling Imprecise Computations*, IEEE Computer 24.

D. Presotto, R. Pike, K. Thompson and H. Trickey [May 1991], *Plan 9, A Distributed System*, Proceedings of the Spring 1991 EurOpen Conference, Tromsø, Norway.

M. Rosenblum and J. K. Ousterhout [October 1991], *The Design and Implementation of a Log-Structured File System*, Proceedings of the 13th Symposium on Operating System Principles, Pacific Grove, CA.

G. J. M. Smit, P. J. M. Havinga and M. J. P. Smit [April 1992], *Rattlesnake: A Network for Real-Time Multimedia Communications*, Proceedings of Multimedia'92, 4th IEEE ComSoc International Workshop on Multimedia Communications, Monterey, CA.