Operating-System Support for Distributed Multimedia

Sape J. Mullender[*] Ian M. Leslie[†] Derek McAuley[†]

Abstract

Multimedia applications place new demands upon processors, networks and operating systems. While some network designers, through ATM for example, hae considered revolutionary approaches to supporting multimedia, the same cannot be said for operating systems designers. Most work is evolutionary in nature attempting to identify additional features that can be added to existing systems to support multimedia. Here we describe the Pegasus project's attempt to buil an operating system from the ground up with multimedia support as a prime objective.

1 Introduction

Since the invention of electronic computers in the forties, every decade has been characterized by new ways in which they were used. In the fifties, people used sign-up sheets to reserve the computer for an hour's work; in the sixties batch processing was introduced; time sharing became pervasive in the seventies; the PC and networking came in the eighties; and now, in the nineties, we see the introduction of multimedia.

These days, every self-respecting computer vendor sells computers with some form of multimedia support. Some workstations now have cameras built into them, PCs come with multimedia applications, even game computers now make use of CD-I. From a research viewpoint, multimedia seems to be a solved problem; can't we see the wonderful demonstrations from every vendor?

We argue that the multimedia applications on most systems today are inflexible, they more or less take over the machine and cannot be combined with other applications.

Multimedia, we claim, is only real if the different media are treated with equal respect. Audio and video should not be second-class media on which the only operations are capture, storage and rendering, but media that can be processed — analysed, filtered, modified — just like text and data. This processing should not be a privilege of dedicated operating-system processes, but should be possible to do, interactively, with ordinary applications.

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

[†]Cambridge University Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, United Kingdom

Figure 1: Architecture of the multimedia workstation

Figure 2: Principle of the ATM Camera

Existing multimedia systems do not have this ability. For example, on typical PC platforms, multimedia applications run in real time but take over the machine; on Unix platforms, multimedia applications co-exist with other applications, but they hardly run in real time. Sometimes, dedicated hardware can capture and render multimedia in real time, but the data is far removed from the processor so that no processing is possible.

The value of audio and video depends critically on the ability both to process and to render them in real time. This is hard. The value of *interactive* audio and video additionally depends on being able to capture, process and render it with fraction-of-a-second end-to-end latency. This is even harder.

In the Pegasus project, groups at the University of Cambridge Computer Laboratory and the University of Twente Faculty of Computer Science are rising to the challenge of providing architectural and operating-system support for distributed multimedia applications.

Pegasus is a European Communities' ESPRIT[1] project which is now halfway through its three-year funding period.

The goal of Pegasus is to create the architecture for a general-purpose distributed multimedia system and build an operating system for it that supports multimedia applications. A few specific applications will be implemented in order to prove the practicality of the system.

The architecture consists of: multimedia workstations; general-purpose and special-purpose multimedia processing servers; a single storage service for all types of data; and Unix boxes as the platform for the non-real-time control part of multimedia applications and applications unrelated to multimedia. All of the components are connected through an ATM network, which provides the bandwidth and can provide latency guarantees for interactive multimedia data. Multimedia capture and rendering devices are connected directly to this network, rather than being connected to, for example, workstation buses. This architecture is explained in Section 2.

The operating system support in Pegasus consists of a microkernel, named *Nemesis*, that supports a single address space with multiple protection domains, and multiple threads in each domain. There is scheduler support for processing multimedia data in real time. Nemesis has a minimal operating-system interface; it does not — at least, not now — have a Unix interface. However, processes on Nemesis can be created, be controlled by, and communicate with, processes on Unix. We expect multimedia applications to consist of symbiotic processes on Nemesis and Unix, where user interface and application control will be provided by the Unix part, and real-time multimedia processing by the Nemesis part. Later, perhaps as part of another project, parts of the Nemesis functionality could be ported to a general-purpose operating system, or a Unix emulation provided over Nemesis. Nemesis is described in Section 3.

System services are viewed as objects: abstract data types accessed through their methods. When invoker and object share a protection domain, method invocation is through procedure call; when they share a machine, and thus an address space, invocation takes place through a protected call, or 'local remote procedure call';

[1]The Pegasus Project is supported by ESPRIT BRA project 6586 and partially supported by the Cambridge Olivetti Research Laboratory and a grant from Digital Equipment Corporation.

when they are on different machines, invocation goes via remote procedure call. Objects are located using a distributed name service. The name space is global only in the sense that every entity, in principle, can name any object in the universe; it is not global in the sense that there is one root to the name space, or that one name identifies the same object anywhere. Each protection domain contains a local name server which maintains connections with name servers elsewhere. The name server assists in establishing the appropriate channels through which local and remote objects are invoked. The name server is described in Section 4.

The Pegasus File Server is a log-structured file service designed to store and retrieve multimedia files in real time and to scale to a very large size. Scaling the file-server design up to terabyte capacity has forced us to redesign the log-structured file-system structures as they occur in Sprite or BSD4.4. The Pegasus File Server uses a buffering and storage strategy that prevents loss of data in case of failure of a single component. The Pegasus File Service is described in Section 5.

2 Systems Architecture

In this section, we will show and explain the unusual architecture of the Pegasus system. The system consists of workstations and servers, interconnected by an ATM network. We use an ATM network as it can provide high bandwidth and low latency. ATM networks can scale gracefully to large sizes and link bandwidths and very large aggregate bandwidths.

Multimedia systems need special hardware for input and output of digital audio and video. Once digitized, video and audio streams must be transported to where they are processed, stored or rendered. Video requires substantial, but not staggering bandwidths: using frame-by-frame compression, for instance with JPEG, a video stream requires no more than a megabyte per second. Modern networks can easily provide this bandwidth. Using compression methods that compress groups of frames, such as MPEG, much higher compression can be reached, albeit at the cost of higher end-to-end latency. Audio has modest bandwidth requirements compared to video, but is much more susceptible to jitter, that is, the irregularities in the transport and processing times.

For smooth and efficient handling of interactive digital audio and video, the paths between origin and destination must be as short as possible. Gratuitous processing and transportation increase the end-to-end latency and hence decrease the quality. Thus, it is desirable that audio and video data are not handled by operating-system and application code except when application-specific processing is being carried out.

Figure 1 shows an important aspect of the Pegasus architecture — the target end-system architecture. The figure shows a conventional workstation and its network interface connected to an ATM switch. However, also connected to the switch we see a camera device, a display device, an audio device, and then the rest of the ATM network. The important point is that the switch is under control of the workstation, that is, all connections through the switch are managed by the workstation, so that the workstation is also in control of the multimedia devices.

This setup is much like that of the *Desk-Area Network* [.hayter 1991.]. However, in a real DAN, an ATM switch fabric actually forms the central backbone of the workstation itself; CPU, memory and devices all communicate via the switch. The Pegasus project, partly because of its time frame of only three years, uses a conventional bus-based architecture for its processor devices, but uses the DAN mechanism for connecting multimedia devices.

In this architecture, when video flows from a camera in one system to a display in another — as is the case in video-phone and video-conferencing applications —

Figure 3: Architecture of the ATM display

no processors need to process any video data. This goes for the audio data too, of course. Hence the processors in the workstations, at both the camera and display, only need to manage the connections and devices.

2.1 Some ATM Devices

This section briefly describes the ATM devices used by the Pegasus project to provide a multimedia platform. More details of the DAN devices are available in ¡.barham:jsac.¿.

The ATM camera [.pratt 1993.], directly produces digital video as a stream of ATM cells. The principle of the ATM camera is schematically depicted in Figure 2. Scan-lines of video are digitized and when eight lines have been buffered, they are encoded as *tiles*, rectangles of 8×8 pixels. A number of tiles are packed into the payload of an AAL5 frame together with a trailer that provides the x and y coordinates of the tiles with respect to the video frame, and a time stamp that identifies the frame that the tile belongs to.

Cameras can be equipped with one or more compression devices. The device to be used is identified when the virtual circuit is established. Currently, both raw video and motion JPEG are supported. Using AAL5 allows interaction with standard AAL5 implementations and offers protection against rendering or decompressing faulty tiles.

The version of the ATM camera now in production also includes audio capture capability.

The ATM display, shown in Figure 3, implements a single primitive, that of displaying arriving pixel tiles on incoming virtual circuits to windows on the screen. The virtual-circuit identifier (VCI) is used as an index into a table of window descriptors; each window descriptor has an x and y offset from the top-left-hand corner of the display, and clipping information. By manipulation of these contexts, a window manager can control which virtual channel, and thus which process, can access the different pixels of the screen.

Incoming data can be coded as compressed or uncompressed tiles. Note that as tiles essentially represent bit-blit operations of fixed size, from the viewpoint of a display, there is a unification of video and graphics. The code in conventional window systems that does the multiplexing of windows to the display can largely disappear; the multiplexing is done via the display's window descriptors. The window manager, exerting its control over the creation and modification of these descriptors, can create windows on screen, move them, resize them, iconize them and raise or lower them. It can also use a window descriptor that allows it to write the whole screen for decorating windows with title bars and resize buttons.

While the hardware for the display is under development, software emulation using a DS5000-25 is being used.

Finally, there is an ATM DSP node which combines digital signal processing and audio input and output. This device contains DACs and ADCs and packs and unpacks audio samples into ATM cells. Each such cell also contains a time stamp.

Our experience so far indicates that ATM devices are simple to construct and that they allow a natural combination of video data and graphic data on a display. The use of *tiles* for video reduces latency in several places from a 'frame time' (33 or 40 ms) to a 'tile time' (30 to 40 μs). Since latencies tend to add up, this is an important reduction.

Figure 4: Pegasus architectural overview

2.2 Control Protocol

Multimedia devices generate two streams of data on two distinct virtual circuits. One is the actual data stream which was cursorily described above. The other is a control stream; this is a bi-directional low-bandwidth stream that is used to control the device and for purposes of synchronization.

Both data and control virtual circuits are established through the normal mechanism of ATM signalling, although in the case of many of the ATM devices, this signalling is handled by a management process on the attached workstation, rather than by the device itself.

Typically, the device manager will connect the data stream directly to the sink or source; however, the control stream would normally be connected to a local synchronization process. For example, a host that wishes to send synchronized audio and video, will do so by having the audio node and camera send the audio and video data streams separately (they have to end up in different devices too, at the other end), while a local process will merge the two control streams into a combined control stream for the playback control process at the rendering end. The playback control process is then responsible for the synchronization of the play-out of the various streams arriving at it, based on the source synchronization information from the remote manager(s) and data arrival events.

The Pegasus File Server, which can also be viewed as a multimedia device in this context, uses the control stream associated with an incoming data stream to generate index information that can later be used to go to specific time offsets into a media file or a set of synchronized files.

2.3 Systems Components

An overview of the Pegasus architecture is shown in Figure 4. In this figure, we can distinguish a Pegasus multimedia workstation, multimedia compute server, storage server and Unix server, all interconnected by an ATM network.

Each site is using locally developed ATM switches to provide the ATM network: the Fairisle switch in Cambridge [.fairisle.], and the Rattlesnake switch in Twente [.smit 1994.].

The architecture of the multimedia workstation is as described above; multimedia input and output devices are connected to a local ATM switch (for which we use the Fairisle switch) and the rest of the workstation is entirely conventional. The multimedia processing nodes do not have special devices attached to them.

The multimedia workstations and processor nodes are controlled by a microkernel, called *Nemesis*. This kernel, which is discussed in more detail in Section 3, provides support for multimedia applications: timely scheduling and efficient interprocess communication.

One or more nodes in Pegasus run Unix. Applications on this platform have access to a rich collection of tools — compilers, text processors, graphics support, etc. — which, due to available effort, we do not intend to make available on the Nemesis kernel. We expect many multimedia applications to be split over Unix and Nemesis; the Unix part will contain the control functionality, whereas the Nemesis part will contain the necessary real-time functionality for audio and video processing.

This separation is entirely inspired by practical considerations. The Pegasus design team does not have the resources to add the kind of scheduling necessary for multimedia processing to existing operating-system platforms, they are too big to modify[2]. Separating Nemesis and Unix gives us the best of both worlds: a testable and measurable platform for multimedia applications and all the functionality of Unix. It is for another project to port Nemesis functionality to Unix or vice versa.

3 Kernel Support

The Nemesis kernel implements several unusual features, some of which are present to aid in the implementation of multimedia applications, others for the simple reason of efficiency and tidiness. Here we summarize the major features.

3.1 Memory Model

A Nemesis kernel provides a number of distinct, schedulable entities, called *domains*. While all domains share the same virtual address space, privacy and protection are implemented using the appropriate access rights in the virtual address translations. Code executing within a domain may access memory within another domain only if both domains have explicitly arranged to share the memory.

Some examples highlight the approach: shared library segments would be mapped readable in every domain; a unidirectional inter-domain communications channel would be mapped read/write in the source and read-only at the sink; objects may be shared in shared read/write segments; etc.

The cost of using a single address space is the penalty of load-time relocation. We try to amortise this cost by caching the results of such relocations and then aim to reload an application at the same virtual address at which it was last executed. In this we are helped by the use of 64-bit VM architectures, which allow a sparse allocation of addresses so that we can arrange reuse with high probability. Consider for example allocating the top 32 address bits of a 64 bit virtual address based on a 32-bit hash function of the code to be executed.

The benefits of a single address space we are aiming for are: simplified sharing of data structures (in particular objects) between domains, and the removal of virtual address aliases which can result in significant context switch costs with caches accessed by virtual address.

3.2 Virtual-Processor Model

A domain differs from the normal concept of a user process in the way in which the processor is presented to it. In the case of a process, the processor is taken away from it by *suspending* it and is returned by *resuming* the process to exactly the state in which it was when it was suspended. This gives the illusion to the process that it is running on its own *virtual processor*; it also hides from the process any information about the current processor availability — the process has no way of knowing *when* it has the processor.

In Nemesis, the processor is taken away from a domain by *deactivating* it; deactivation involves storing the state of the processor into a data structure shared by the kernel and domain, the Domain Information Block. When the domain is next scheduled, the processor is given to a domain by *activating* the domain; activation involves transferring execution to an address specified in the activation vector entry in the Domain Information Block.

For a domain supporting a traditional single-threaded model of execution, activation start up code would just restore the saved context and the user code would continue to execute. Another common use within the Pegasus project would be for

[2]Yes, even Mach 3.0.

the entry point to be a user-level thread scheduler. In this case the mechanism provides functionality similar to *scheduler activations* [.anderson sosp13 activations.]. Finally, some domains may be completely event driven, for example, device driver domains.

Hence it is simple to support the standard programming models on this activation model; in fact all operating systems do it, but it is usually the case that the asynchronous nature of interrupts and rescheduling events is hidden from the user level code.

The Nemesis mechanism provides a number of advantages for the types of multimedia applications we are considering. First, it provides a means of informing applications when they have the processor; a user-level scheduler can use this information, together with the current time, to make more informed decisions about the fate of the threads which it controls. Second, because thread scheduling is performed by the application, the user-level scheduler has direct control over the behaviour of its threads, and does not have to resort to describing their behaviour to a central scheduler in terms of priorities and deadlines. Third, once a domain is given the processor, it keeps it until its time quantum expires, or it voluntarily yields the processor because it has no more work to do. This avoids the problems encountered in kernel level thread implementations when threads block in the kernel and the kernel scheduler gives the processor which was running the blocked thread to a thread belonging to another process. Nemesis has no blocking system calls except "suspend" which will typically only be called by a domain user-level thread scheduler.

3.3 Domain Scheduling

To explain the scheduling mechanism adopted in Nemesis requires an understanding of how we see a flexible multimedia platform being used. The allocation of resources to applications will not be controlled soley by the applications themselves. Rather we see users being able to control processor allocation much in the same way that they control pixel allocation in window systems. Thus applications will not always get what they want; they will have to adapt to the resources they are given. However, for a particular time, seconds or tens of seconds, some of the resources given to an application may be viewed as "guaranteed". The application may choose to use an particular algorithm on the basis of this guarantee. It may also be able to exploit unguaranteed resources which become available fortuitously.

The approach to scheduling in Nemesis is to schedule domains with a weighted scheduling discipline, where the weights are calculated from the user's current policy. Within a given time frame, not all domains may use their allocation; the policy for sharing out remaining resources is still the subject of investigation. While domains have some processor allocation remaining, the current scheduler implementation uses an earliest deadline first algorithm to select between them.

Above this primitive-level scheduler, and running on a longer time scale is a *Quality-of-Service*-manager domain whose task is to update the scheduler weights; this is performed not only in response to applications entering or leaving the system, but also adaptively as applications modify their behaviour — this is performed on a longer time scale that the individual scheduling decisions in order to smooth out short-term variations in load.

3.4 Events

Nemesis provides a single mechanism by which domains can communicate the occurrence of *events* to each other — this also includes indications from interrupt handlers. A domain is eligible for scheduling when it has pending events, at which

```
... <unprivileged code>

begin_KPS();        /* enter privileged section */
TRY
        ... <privileged code>

FINALLY
end_KPS();          /* leave privileged section */
END;

... <unprivileged code>
```

Figure 5: Coding a Kernel-Privileged Section

point it is included in the scheduling mechanism described above. Then, when a domain is activated, it is informed of pending events.

Events themselves do not carry values, but merely indicate that something has occurred. This may be the updating of a shared object, the arrival of a message from the network, passage of time, etc.; however, closures (ie. methods and data) are associated with each event and hide this heterogeneity from the event dispatcher.

The examples of a protocol domain processing arriving packets and inter-domain procedure calls highlight the need for two types of event signalling: synchronous and asynchronous, depending on whether signalling an event should cause a domain to voluntarily give up the processor to the signalled domain or continue executing. In the inter-domain call example, implemented using a pair of message queues in shared memory between the relevant client and server domains and a pair of events, lowest latency for a client/server interaction will be achieved by the client and server implementing the synchronous form of notification. However, a domain performing demultiplexing of incoming packets may be most efficient using the asynchronous means.

3.5 Kernel Privileged Sections

Device drivers and other trusted modules need to be able to protect themselves against interrupts, have access to privileged instructions, etc., for some part of their operation. The code that requires this access is often a tiny proportion of the total module; however, most operating systems would require that the whole module run in kernel mode, whether linked statically or dynamically loaded. Furthermore, it becomes a property of the code that it runs in kernel mode, rather than the data the code is manipulating.

Nemesis offers the concept of the Kernel-Privileged Section to meet the requirement for a dynamic and extensible means to provide access to kernel mode. Privileged domains may define sections of their code which need to be executed in kernel mode. In a block-structured language this would naturally be a basic block enclosed with some form of `TRY ... FINALLY` construct allowing privileged code to raise exceptions but forcing the thread to leave kernel mode before any handler outside the privileged section is invoked (see Figure 5). The implementation of the Kernel-Privileged Section (i.e. the `begin_KPS` and `end_KPS`) is highly processor dependent — on 68k, MIPS and ARM processors it leads to various traps implemented in a non-procedural manner, while the aim on the Alpha is to implement a PAL instruction to achieve the desired effect.

In many ways the Kernel-Privileged Section idea is akin to using locked critical sections for currency control, whereas most other operating systems have a model of kernel mode access more akin to *monitored procedures*.

3.6 Nemesis State

A primitive form of the Nemesis kernel, *Nematode*, has been implemented on DECstation 5000 [.hyden:thesis.]; this provides domains, events, and scheduling support. Currently Nematode is being evolved to conform to the machine independent interfaces defined for the Nemesis kernel.

The VM model and communications abstraction are adopted from those used for Wanda [.dixon:thesis.]; migration of this code awaits completion of the Nemesis kernel.

4 Naming and Invocation

Most objects (entities, things) will be used locally. Therefore, most names of objects used will be names of local objects. Name resolution should, therefore, be most efficient for local names. This implies that local names should be shortest and suggests that names of local objects should normally be near to the root of the naming tree.

This, it must be clear, is a deviation from a trend towards using *global name spaces*. In a singly rooted global name space, the shortest path names refer to countries or organizations; it is rare that we wish to name those by themselves. The most widely claimed advantage of a global name space is that objects have the same name anywhere and that this facilitates sharing. What actually facilitates sharing much more is the proper use of naming conventions: One can often *guess* somebody's electronic-mail address, one looks for TEX macro files in subdirectories of /usr/local/lib or /usr/lib, one gives C source code files a '.c' extension. If the conventions are disobeyed, programs fail.

By using naming conventions properly, one can create name spaces that are only global in the sense that any object anywhere can be named, but not necessarily by the same name everywhere. The root of the naming tree can be the most local object and longer path names generally name objects further away. Conventions must be used to allow object sharing and there is no reason why one convention could not be the use of a subtree named /global for global names.

This sort of naming is used in Plan 9 from Bell Labs. ¡.pike name spaces plan 1993.¿ have already put forward some of the arguments for naming conventions being more important than global name spaces. Our naming mechanisms have been heavily inspired by those of Plan 9 as shall become clear.

Every process starts up with a built-in name space. Usually, this name space is inherited from a parent process and is at least partly shared with other name spaces. The name space consists of a *local name space* which names objects local to the process, and *mounted name spaces* which name objects external to the process. The mount point of a mounted name space is a local object with a connection to a name space in another process. Name resolution in mounted name spaces takes place by making name-lookup requests through the connection to the other process. The result of this resolution is an object handle.

Using an object handle, objects can be accessed through their *methods*. The precise manner in which methods are invoked depends upon the "domain relation" between invoker and object. If they share a protection domain then the invocation is a procedure call; when they are in the same address space but different protection domains (for example on the same Nemesis machine) invocation is by protected call;

and when in different address spaces invocation is performed by remote procedure call.

When making an invocation there is always code at the invoker's end that depends on the call interface. In the case of a local procedure call, this interface-dependent code is generated by the compiler. In the case of system calls it is loaded from a library and in the case of remote procedure call it is generated by a *stub compiler* and linked with the rest of the caller's code.

Client stubs for far-away objects may do more than just transport call parameters to the remote objects; they may, for instance, perform caching so that there is no longer a one-to-one mapping between client calls to the stubs and calls to the remote objects. Such intelligent stubs are referred to as *agents* or *clerks*.

When objects can migrate, for instance, to where they are accessed, the interfaces to them may change. This means that the interface with which calls are to be made is not always known *a priori*; the calling code depends on where the object is found when it is invoked.

Early distributed systems solved this by using the most general invocation method always: remote procedure call. This is not an optimal solution, especially now that dynamic linking can be used to invoke optimal code for the kind of call to be made in the case at hand.

An object-naming mechanism can be used to make the mechanism whereby object-interface code is loaded transparent. In our model, the resolution of the name of an object results in a *handle*. This handle is essentially a pointer to the interface to the object. For our handles we use *maillons* [.maisonneuve references as chains of links.], which consist of an opaque, fixed-size, object reference and a pointer to a function that returns the address of the interface when called with the reference as argument. The extra level of indirection provided by the maillon allows connections to objects to be set up, or objects to be fetched before their first invocation, but in the most common case — the object is already there and ready to be invoked — the maillon imposes very little overhead.

Object handles are first-class objects in that they can be passed as arguments in local and remote procedures. Passing an object handle for a local object to a remote process has the side effect of creating a connection through which the object can be invoked remotely.

The Pegasus remote-procedure-call mechanism is based on ANSA's RPC and layered on MSNA [.mcauley phd.]. The Multi-Service Network Architecture is a protocol hierarchy for ATM networks that also caters for continuous-media transport.

5 Storage

The storage system in Pegasus is intended to store traditional file data as well as multimedia data efficiently. A storage service for multimedia data must have a large storage capacity (video produces half a megabyte per second compressed, so a half-hour video already occupies a gigabyte) and a guaranteed (fixed) service rate.

Ordinary data usually occupies less space and does not require a guaranteed service rate. The data rate does not have to be constant, but should be as high as possible. Locality of reference can be exploited by caching data in client and/or server memory. Most modern file systems demonstrate that caching yields substantial performance gains.

This applies to naming data too, albeit that directories can be cached more effectively when the semantics of directory operations are exploited in the caching algorithms.

In contrast, caching video and audio is usually not a good idea. If the system can already guarantee the appropriate rate for a video or audio stream when it is not cached, caching it will only use up memory, but cannot result in a higher performance — a fixed performance is desired. To make matters worse, caching would often be counterproductive: Most video sequences and many audio sequences are larger than the cache, so, by the time a user has seen, or an application has processed, a video to the end, the beginning has already been evicted from the (LRU) cache.

Since different kinds of data require different treatment in our storage service, it was decided to make a hierarchical design for it, where a common bottom layer is responsible for reading and writing the data on secondary and tertiary storage devices and maintaining the storage structures on them. Above this layer, different service stacks can be built using specialized algorithms for particular kinds of data.

These service stacks can be partially or wholly mirrored in file-server agents on client machines. Thus, caching strategies, for instance, can be jointly implemented by corresponding layers of code in client and server machines.

The service stack for continuous data on the server has been designed to interact directly with the multimedia devices of Pegasus. As described in Section 2, continuous streams composed of several substreams (synchronized video and audio is a typical example) will cause several data streams and one control stream to be generated. The storage server stores the data streams and uses the control stream to generate indexing information. This information then allows reading synchronized streams from a particular point, and fast forward, reverse play, etc. of these streams.

The bottom layer of the Pegasus storage service is called the *core layer*. It manages storage structures on secondary and tertiary storage devices and carries out the actual I/O. Pegasus uses a log-structured storage layout as was exemplified by Sprite LFS [.rosenblum sosp13.].

The log is segmented in megabyte segments. Each segment is striped across four disks. A fifth disk is used as a parity disk and allows recovery from disk errors.

Normal file data ends up in the log similarly to Sprite LFS. Continuous data, however, is collected in separate segments, although their metadata (the *inodes* or *pnodes* as we call them) are appended to the normal log.

The speeds of modern disks are such that the overhead of seeks between reading and writing whole segments is less than ten per cent, so that a transfer rate of at least five megabytes per second per disk is possible on high-performance disk hardware. Striping over four disks makes a total bandwidth of 20 MB per second possible. We have not been able to test this yet, since our ATM network runs only at a mere 100 megabits per second, just over 10 MB per second.

Partly as a consequence of storing multimedia data, we have to expect that our storage service will grow large. We have set ourselves the goal to make the storage-service algorithms scale to a system size of 10 terabytes. Cleaning[3] algorithms for a storage service of this size have to be designed carefully. If any part of the cleaning process scales with, say, the square of the system size, cleaning a terabyte file system will take a very long time.

We are currently implementing a cleaning algorithm whose complexity only depends on the number of segments to be cleaned and the amount of 'garbage'. Roughly, it works as follows. During normal operation of the file system, the core maintains a *garbage file*. Every time a client write or delete operation creates garbage, an entry describing the hole in the log that corresponds to the obsolete data is appended to the garbage file.

[3] *Cleaning* in a log-structured filing system is the act of recovering space which holds out of date information. Information may become out of date either because a later copy has been written or it has been logically deleted.

When the file system needs to be cleaned, the garbage file is read and its entries are sorted by segment number. Then, a single pass through the garbage file is needed to find and clean all segments containing garbage. When cleaning is complete, the garbage file is truncated to a single entry describing the old garbage file itself.

Allowing client operations to continue during cleaning does not complicate the cleaning algorithm. At the start of a cleaning operation, the current place in the garbage file must be marked and cleaning uses only information before the marker while new garbage is appended after it. When cleaning is complete, the portion of the garbage file before the marker is deleted.

The first prototype of the core of the Pegasus file server now runs, with an incomplete cleaning mechanism. Higher-level services are being added; a Unix *v-node* interface is installed which allows the storage system to be used as a Unix file system.

Since files are stored on RAID, recovery from disk failures is straightforward. Once files have reached the disk, it is unlikely that they will be lost in a crash. Files, therefore, should be put on disk as soon as possible after they are written by the application. However, from a performance viewlpqoint, disk writes should be delayed so that overwrite operations and delete operations can be exploited to save disk operations. In the Pegasus storage service we have tried to get the best of both worlds.

For this, we make use of the assumption that client and server machines crash independently. When an application makes a write operation, the client agent sends the data to the server and keeps a copy of the data in its buffers. When the server receives the data, it acknowledges this to the client agent which, in turn, unblocks the application. The data is now safe under single-point failures: when the server crashes, the client agent notices and either writes the data to an alternative server or waits for the crashed server to come back up; when the client machine crashes, the server will complete the write operation.

When there is a power failure, client and server will crash together. To guard against this, the servers can either be equipped with battery-backed-up memory, or with an uninterruptible power supply. With the latter, when a power failure occurs, the server has time to write its volatile-memory buffers to disk and halt.

These mechanisms obviate the need for writing data to disk quickly. For normal file traffic, this is not only beneficial for write performance — ¡.baker sosp13.¿ showed that 70% of files are deleted or overwritten within 30 seconds — but also for cleaning performance: The data that does eventually get written to the log is reasonably stable, so garbage is created at a much lower rate.

6 Conclusions

The Pegasus project reflects our belief that if distributed multimedia is to be supported effectively, a holistic approach to system design is required. Multimedia is not just a bolt on; it requires a fundamental reexamination of most aspects of the infrastructure. We have thought carefully about integrating multimedia devices into the network architecture of the system, we have looked at the data paths from camera lens to display screens, and we have analysed storage infrastructures from a performance, reliability and consistency perspective.

Thus far we have found that this approach gives a clean system design and makes our implementations efficient and simple. The desk-area network as the connecting infrastructure for machines and devices has greatly simplified the architecture of the rest of the system.

In the storage service, we have discovered that techniques for consistent caching, data buffering, log structure and RAID, each of which, by itself, is difficult to integrate in an existing environment, can be combined in a new storage system architecture. Consistent caching, buffering and RAID gave us reliability (no data loss in a single crash); log structure and RAID give us good write performance.

Pegasus is only half-way through its funding period now and a lot of work still needs to be done. We hope we can demonstrate a complete system in two years' time. The results of our project are naturally public and we intend to make all code available where it is not restricted by licences from others.

7 Acknowledgements

The work presented here is the work of a team whose members we gratefully acknowledge: Paul Barham, Ralph Beckett, Richard Black, Peter Bosch, Tatjana Burkow, Shaw Chaung, Simon Crosby, Feico Dillema, Richard Earnshaw, Dave Evers, Robin Fairbairns, Daniel Gordon, Mark Hayter, Paul Havinga, Eoin Hyden, Pierre Jansen, George Linnenbank, George Neville-Neil, Ian Pratt, Timothy Roscoe, Paul Sijben, Gerard Smit, Martijn van der Valk, Lars Vognild, and Lex Warners.

8 References

.▯