Design and Implementation of a Real-Time ATM-Based Protocol Server

Martin Borriss Hermann Härtig
Dresden University of Technology
Operating Systems Group
{borriss,haertig}@os.inf.tu-dresden.de

Abstract

This paper describes design and implementation of L^4 ATM, an ATM (Asynchronous Transfer Mode) based networking server. While ATM emphasizes deterministic high-speed communication, applications can not yet fully utilize its potential. We demonstrate an architecture—and a corresponding implementation—to resolve this dilemma by developing implementable resource quantification techniques and QoS (Quality Of Service) management algorithms for host resources.

L^4 ATM has been built in the context of DROPS *(Dresden Real-Time OPerating System).* DROPS *supports coexisting real-time and time-sharing applications in a μkernel environment.*

Evaluating L^4 ATM's implementation in a real-world environment, we show that (i) performance guarantees are maintained under heavy time-sharing load, and (ii) the implementation outperforms a standard OS significantly.

1. Introduction

This section introduces the DROPS architecture [12] and states design criteria applying to all sub-projects within the DROPS context.

1.1. Motivation

Applications with real-time requirements, such as multimedia applications, can be supported by end systems in two contradictory ways: (1) Use standard (e.g., non-real-time capable) operating systems software on hosts equipped with vast amounts of resources to account for the worst case; (2) Alternatively, use real-time-aware operating system software, thus providing performance guarantees to real-time applications while using less powerful hardware compared to the first approach.

This research was supported in part by the Deutsche Forschungsgemeinschaft (DFG) through the Sonderforschungsbereich 358.

Firstly, we believe that the second alternative is technologically more desirable. Secondly, we feel that—although complex—building such a system is attainable and practical. This observation has guided the design of DROPS. The remainder of Section 1 reviews DROPS' essentials.

1.2. DROPS architecture

Based on the L4 μkernel [17], DROPS incorporates multiple OS personalities: Standard time-sharing tasks use the L^4Linux server [13], a port of the monolithic Linux kernel to the L4 μkernel. In contrast, real-time applications are free to use dedicated real-time servers for predictable performance. Drivers run in user space. Resource manager components are responsible for enforcing and monitoring host resource reservation, such as buffer space, processing times, cache utilization and I/O-bandwidth. Figure 1 shows the system architecture.

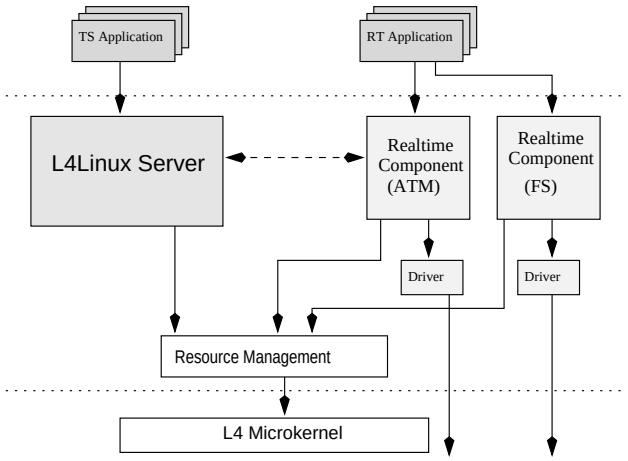

Figure 1. Simplified DROPS architecture.

The L^4Linux server provides the environment for standard time-sharing applications and buys us (1) support for

a broad range of hardware devices, and (2) binary compatibility to Linux. With the exception of the Linux scheduler, only architecture-dependent parts needed to be modified to adapt Linux to the L4-μkernel. System call traps are redirected via the L4 μkernel (*trampoline mechanism*). Hardware interrupts are translated into IPC messages to dedicated threads. To control interrupt latency, protection of critical regions within the L^{4}Linux server is done by using explicit synchronization instead of blocking interrupts. Host memory is divided at boot time between the L^{4}Linux server and real-time components, taking cache access characteristics into account [10].

1.3. Real-time model

Similar to the *path abstraction* in the experimental Scout [21] system and to the concept of *compound sessions* by Anderson's LBAP model [4], real-time components can be lined up in chains. The DROPS model is based on jitter constrained streams [9], an abstraction on numerous parameter sets (such as the QoS parameters defined for ATM networks [23]). The model develops quantitative techniques for resource management, for example, computation of required buffer space to compensate for jitter (Section 2.3).

The data model builds on L4' virtual memory management concept of *flexpage* mapping operations [11], resembling *fbufs* [7], but extending it with the capability of precisely timed revoke operations of page mappings.

Each real time component must provide the uniform DROPS interface, but is free to offer additional interfaces.

Scheduling in the L4 μkernel uses static priorities. On top of that, the model provides a flexible periodic scheduling framework. Applications may reserve a certain priority level within a given interval for a number of cycles. For the reserved cycles, priority of the application will be temporarily raised. Time-sharing processes (that is, low-priority processes) are granted unused or remaining cycles. A call-back mechanism provides feedback to applications, allowing them to scale their requirements as appropriate.

The remainder of this paper discusses design, implementation and performance of one such real-time component, the L^{4}ATM protocol server.

2. L^{4}ATM design

While network hardware support is necessary to offer QoS guarantees to applications, it is not sufficient. To guarantee tight bounds on throughput, delay and jitter, operating system support is mandatory. Operating system and host protocol implementations must provide mechanisms for the management of local resources necessary to pass performance guarantees on to application level. We identified relevant host resources for protocol processing, in particular: CPU time, host buffer space, network bandwidth, cache [18], I/O-bandwidth and memory bandwidth. The machine-global resource manager enforces reservation. Additionally, time-sharing applications are conceptually integrated into the resource management design to allow strict separation from guaranteed services.

Requirements. Applications using L^{4}ATM require: (1) Guaranteed throughput; (2) Bounded latency; and (3) Bounded jitter. These requirements are expressed in terms of a traffic specification and mapped to actual resource reservation by the host (within L^{4}ATM and driver) and by the network (within ATM switches).

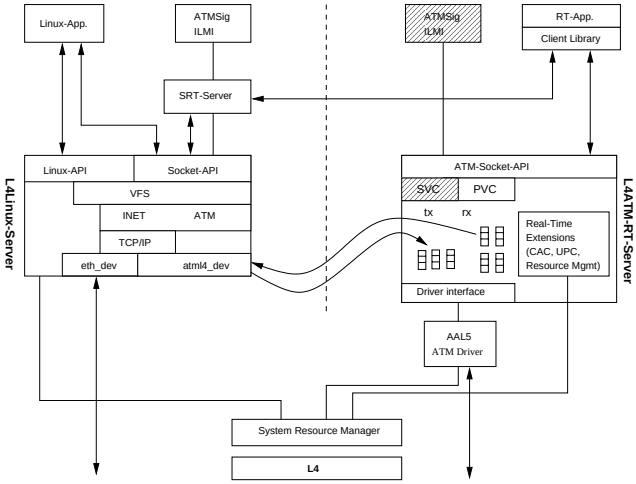

Figure 2. Integration of the L^{4}ATM protocol server into the DROPS framework.

For practical reasons we rejected the full implementation of heavy-weight transport protocols in favor of a native ATM protocol stack. Figure 2 shows the overall design and its integration into the DROPS framework. Note that API calls unsupported by real-time servers are transparently redirected to L^{4}Linux by the client library and a "SRT server" task. ATM signalling is performed by a Linux daemon, which is part of the Linux-ATM distribution [3].

2.1. Process structure

For L^{4}ATM, we chose a multi-threaded design. ATM connections are associated with dedicated *worker threads*. Using BSD-socket terminology, worker threads handle `read()` and `write()` calls. Additionally, traffic management algorithms (with the exception of admission control) run in worker thread context.

A *service thread* handles non-time-critical requests, such as connection establishment, admission control, and connection tear-down.

Finally, a *pseudo interrupt thread* takes incoming protocol data units from the hardware driver, buffers messages if necessary and triggers activation of a worker thread blocking on this connection.

The resulting concurrent server is scalable and fits well to L4's synchronous IPC. Most importantly, however, an efficient implementation of real-time policies is possible (i.e., per-connection priorities and L^{4}ATM's *thread suspension* technique used for enforcing resource reservation). Thread support—conforming to POSIX threads—for L^{4}ATM is contained in the platform dependent layer, providing an 1:1 mapping of pthreads to L4's kernel threads. Synchronization primitives (`mutexes`) support priority inheritance to avoid priority inversion.

2.2. Data transfer

To achieve low communication latencies and preserve the high bandwidth offered by network hardware, our design reduces data copying costs as much as possible without compromising protocol functionality and system integrity.

Transmit data path. Traditionally, the data path looks as follows: Application data is copied into protocol memory. Protocol buffers are then copied by the network interface card into the internal send FIFO and are transmitted. No CPU involvement is necessary for the last data copy (DMA). Thus, `write()` calls are possible with a single CPU-initiated copy. Additionally, L^{4}ATM offers another, zero-copy, interface. Using the page re-mapping scheme, the copy operation between application and L^{4}ATM is avoided and replaced by a temporary or permanent mapping operation (Figure 3).

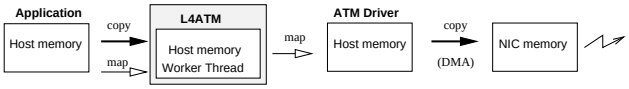

Figure 3. Data path for `write()`.

Receive data path. L^{4}ATM employs both a two-copy architecture (for the standard `read()`-interface) and a single-copy architecture (for the real-time interface). As can be seen from Figure 4, `read()` calls require an extra data copy. Typically, a receive interrupt activity has to identify the higher-level protocol of a PDU, allocate buffer memory, copy the PDU into the protocol buffer and return the old buffer to the ATM device driver. In addition, receive control flow is divided into the pseudo interrupt activity and the `read()` activity initiated by the receiving application, which blocks until data arrives. It is feasible to get rid of this last CPU-initiated data copy, even without special support by the ATM board. However, explicit synchronization and feedback are necessary to signal possible buffer re-use to the hardware driver, requiring knowledge and cooperation by the application.

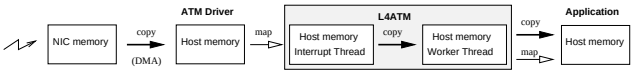

Figure 4. Data path for `read()`.

L^{4}ATM uses linear buffers, similar to Linux' `sk_buffs`. For the single-copy and zero-copy variants, protocol buffers carry only a reference to the actual user buffer.

2.3. Real-time extensions

Real-time behavior is implemented by inserting a set of traffic management algorithms into L^{4}ATM, which implement functionality to enforce meeting of QoS objectives of connections compliant to their traffic specification.

Based on resource requirements computed from the specified traffic parameters, *Connection Admission Control* decides on feasibility of new guaranteed connections, maintains resource utilization statistics and triggers actual resource reservation. *Usage Parameter Control (UPC)* detects non-conforming connections and triggers the local policing function. Note that UPC applies to both real-time and non-real time traffic. Non-conforming connections (i.e., connections exceeding their reserved bandwidth) are subject to policing. The *policing function* provides flow control by restricting the data rate.

Traffic parameters. To express bandwidth and jitter requirements, applications submit a parameter set to the protocol, using the `setsockopt()` call. The parameter set contains: (1) A qualitative description of the requested service, expressed in terms of ATM traffic classes. ATM potentially supports Constant Bit Rate (CBR) service, real-time Variable Bit Rate (rt-VBR) service and a number of non-real-time service classes. (2) Maximum data rate. (3) Sustainable data rate. This is intended for support of staggering streams and maps to the rt-VBR traffic class. For streams with constant data rate, both rates are equal. (4) Maximum service data unit length. (5) Delay variation tolerance (DTV). Applications announce their sensibility regarding jitter.

The purpose of this parameter set is twofold: Firstly, L^{4}ATM performs local admission control, triggers resource reservation and initializes local management information used for the UPC algorithm. Secondly, the same parameter set is signalled to the network to be used for ATM-level admission control and resource allocation.

Integration of time-sharing applications. Applications without guaranteed performance potentially suffer from two handicaps. First, real-time connections use higher priorities than all non-real-time connections, controlling latency. Note that, due to the use of the priority inheritance protocol for mutexes, waiting time at critical sections for high-priority worker threads is bounded. Second, non-real-time connections share the remaining unreserved bandwidth, which is dynamically adapted on start and termination of guaranteed connections. For non-real-time traffic a minimum amount of host resources must be available to allow at least discarding data.

Design of traffic management algorithms is discussed next, implementation issues are described in Section 3.

2.4. Admission control

Admission control for the protocol is designed to manage the resources CPU, buffer space and network bandwidth. From application-supplied parameters and parameters configured statically (such as maximum available network bandwidth) and dynamically (such as CPU speed) required CPU time and parameters for UPC are determined. In case a new connection cannot be accommodated, admission control recommends a modified traffic parameter set to the client, in order to ease re-negotiation.

Buffer space. Buffer space needed for a connection is computed from the following traffic parameters: For transmitting real-time connections, transmit buffers are sized according to the maximum service data unit. For real-time receiving, receive buffers sizes are determined from both bandwidth parameters and the jitter tolerance specified by the client, as can be seen from the following example.

Using the generalized framework developed in [9]: Let $\mathcal{D}$ be the minimum inter-arrival time of two conforming packets, defined by the reciprocal of the maximum data rate: $D = \frac{1}{R_{max}}$. Let $\mathcal{T}$ be the average inter-arrival time of two conforming packets, defined by the reciprocal of the sustainable data rate: $T = \frac{1}{R_{avg}}$. Further, let τ be the maximum acceptable delay variation, $\mathcal{L}$ be the maximum burst length and $\mathcal{P}$ be the buffer size. Then, the following holds [1]:

$$\mathcal{L} = 1 + \left\lfloor \frac{\tau}{\mathcal{T} - \mathcal{D}} \right\rfloor \quad (1)$$

$$\mathcal{P} = \left\lceil \frac{\tau}{\mathcal{T}} \right\rceil \quad (2)$$

As an example, consider a request which adheres to the following traffic parameters: $R_{max} = 125Mbps$, $R_{avg} = 93.75Mbps$, a data size of $max\_sdu = 8192$ bytes and a delay variation tolerance of $\tau = 2ms$. Plugging in the numbers, Equations 1 and 2 yield a maximum burst size of $\mathcal{L} = 13$ packets and a required buffer size $\mathcal{P} = 24$ KB.

L^{4}ATM also maintains network bandwidth utilization. Contrary to network devices, L^{4}ATM has knowledge of the locally achievable throughput. Otherwise, considering a fast network interface, networking hardware may accept bandwidth reservations which cannot be preserved by host systems. Furthermore, this concept allows bandwidth management in environments not supporting network level bandwidth negotiation, for instance due to limited support in network interface cards or in a statically configured ATM environment (using Permanent Virtual Circuits (PVCs) where no dynamic QoS negotiation takes place).

2.5. Usage parameter control and policing

Both algorithms are closely related and are in practice often implemented together. While admission control ensures that enough resources are available for this connection, UPC is responsible for detection of non-conforming connections, that is, connections exceeding their requested bandwidth. After detection of non-conforming traffic policing takes over and guarantees isolation of non-conforming traffic. UPC and policing are always performed in worker thread context.

Packet-based UPC requires a granularity which we find too fine to be implemented in practice.[2] Therefore, L^{4}ATM adopts a mechanism comparable to a token bucket scheme. Each real-time connection has a control interval associated with it. At the beginning of each interval, the connection receives new credit[3]. When transferring data, connections are conforming to their traffic specification as long as enough credits for the current PDU are left. If not, the policing function, which is described below, takes over. Note that this credit-based scheme behaves oblivious. That is, connections do not receive additional credit for using less than its reserved share in earlier intervals.

Worker threads exceeding their announced peak rate are suspended until the start of the next interval (at that point the connection receives new credits). In combination with a synchronous API, this design enforces flow control.

Finding the optimum interval length involves a tradeoff between accuracy of rate control and burst size ("clumping" of packets). Long intervals yield more exact rate control for bursty streams and induce less overhead for the UPC algorithm itself. On the contrary, short intervals bound the maximum burst size and have a smoothing effect on the data

[1] We have to refer to [9] for a complete treatment.

[2] For example, when requesting an 80 Mbps stream consisting of 8 KB-sized packets, average inter-arrival time for a single packet is 0.78 ms.

[3] Credits and intervals are direction-specific, that is, connections may maintain separate credits and intervals for both directions, provided that real-time service has been requested for both directions.

stream. Figure 5 illustrates this for a connection which has 50% reserved bandwidth.

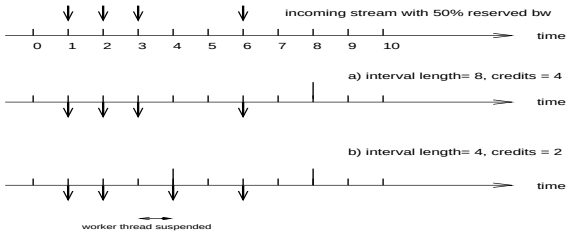

Figure 5. Influence of interval length.

It can be seen that for an asynchronous interface (e.g., the DROPS stream model and the receive direction from the network) short intervals potentially lead to dismission of packets considered conforming when using longer intervals[4]. In practice, very short intervals additionally suffer from rounding errors.

2.6. Driver interaction

This section closes by describing protocol interaction with the driver. The hardware driver provides two IPC interfaces, a send interface and a receive interface. Current design and implementation uses a synchronous send interface, which eases transmit buffer management due to its implicit feedback. That is, using page re-mapping techniques, the application's buffer (or any other producer's buffer) is transmitted to the driver interface, where the ATM board performs AAL processing (e.g., segmentation and checksumming) and transfers the data via DMA into its transmit FIFO queue. On return from this operation, the application's buffer can be recycled safely.

On data receive, the ATM board reassembles the PDU and copies it (via DMA) into host-resident buffers. These buffers are handed to the driver by the protocol on startup. The board triggers an interrupt, causing the ATM driver's receive thread to transmit a reference to this buffer. The pseudo-interrupt thread within the protocol classifies the PDU, physically copies the buffer into a connection-specific buffer and calls `wake_up()` on this connection.

Driver real-time support. To allow for separation of real-time from best-effort traffic, the driver uses two receive buffer pools. The protocol conveys knowledge about the nature (guaranteed or best effort) of a new connection to the driver when opening a connection. This allows for early discard of non-guaranteed traffic on system overload induced by arriving ATM traffic. While the protocol has more information available for making competent QoS decisions, consistent real-time support on all system level maps well into a general DROPS paradigm. Particularly, on hosts with fast networking hardware attached, real-time support already at the driver level is imperative.

While the above scheme is not yet implemented in the driver, the Linux version of the PCA-200E driver already supports the simple cell rate control mechanism offered by the hardware.

[4] In the example, assume the packet at time 6 arrives early at time 4. Depending on available buffer space, this packet could be dismissed in scenario b), but not in scenario a).

2.7. Application interfaces

The interface provided by L^{4}ATM is based on L4's synchronous IPC. Two principal application interfaces are included, which are encapsulated by a library.

Socket Interface. To ease usage of L^{4}ATM and—because the synchronous nature of most socket-level calls maps well to L4's IPC—a BSD-socket compatible interface is provided. Initial test applications and applications ported to native L4 are expected to primarily use this interface. The interface handles both real-time and time-sharing services. Users convey traffic parameters via the `setsockopt()` call, similar to the design of Linux' ATM API [2]. It is important to note that no QoS-related knowledge (except the traffic spec) is needed by applications. The API is extended by calls where user↔L^{4}ATM data transfer is performed via map operations. All measurements in Section 4 have been performed using this interface.

DROPS Stream Interface. To allow interoperability with other DROPS components, the design contains an asynchronous interface based on timed page mapping and page unmapping operations. No explicit flow control is provided here, applications are expected to be able to reserve their resources and conform to their traffic specification. Implementation of this interface is yet incomplete.

L^{4}Linux Interface. As seen by L^{4}Linux server, L^{4}ATM behaves like a device driver. Data is transmitted via a worker thread. For receive, no buffering of Linux traffic is performed in L^{4}ATM, since L^{4}Linux handles buffering itself.

3. Implementation

Implementation of L^{4}ATM's design is fairly complete. This section reviews selected aspects and mentions remaining discrepancies compared to the design.

Traffic management algorithms work well and exact (see Section 4). Static ATM connections (PVCs) are fully functional. Transparent usage by the L^{4}Linux server (including SVCs, IP encapsulation) is stable. Support for DROPS-type chains is yet untested. Driver real-time mechanisms are implemented at the moment.

Hardware. Development platform for the DROPS project are Intel PCs. Testing and measurements (Section 4) were performed on off-the-shelf Pentium-90 machines [5] and Pentium Pro-200 machines[6]. All machines included FORE PCA-200E PCI network interface adapters. The boards support a line rate of 155.52 Mbps and are capable of AAL5 processing in hardware and bus-master DMA transfer. The machines are physically connected to a FORE ASX-200WG ATM switch using OC-3 optical fiber.

Software. The ATM device driver for Linux has been developed in our group and is freely available [5]. The L^4 ATM version is a port of the Linux driver [6].

Protocol implementation is based on the ATM suite for Linux [3]. As a first step, the Linux implementation has been moved to user level [24]. The L4-specific platform-dependent layer of L^4 ATM is built using an early version of the OSKit distribution of the University of Utah [8], a port of the Linux Pthread implementation and a library rebuilding some of typical Linux' kernel functionality (e.g., `sleep_on()` and `wake_up()`) with L4 primitives.

3.1. Resource management

Until we attain more detailed experience using the real-time components a few simplifying assumptions are used in the implementation. Resource amounts are assumed to be linear[7]. In particular, this applies to network bandwidth, buffer space and processor as resources.

Due to requirements for DMA-able buffers (i.e., contiguous physical memory) cache-partitioning schemes are not applied yet.

For simplicity, admission control manages utilization of resources independently. We use known dependency relations of resources. For instance, PCI bus and memory bus usage is strongly correlated with CPU usage. Therefore—lacking better techniques—strong dependencies among resources allow a reasonable approximation in practice.

The assumptions made reduce the number of resources considered in the implementation to CPU time, network bandwidth and buffer space.

CPU time. Machine speed is measured at protocol initialization time. Average cycles needed for protocol and driver processing have been measured. The implementation bases admission control on linear CPU utilization. Measured values for maximum achievable transmit and receive bandwidth are the basis for CPU resource management. For example, a slow P-90 machine has peak transmit rates of about 133 Mbps and peak receive rates of about 90 Mbps in the current non-optimized implementation. Increase in processing time for small packets is compensated for by introduction of a scaling factor λ. Equation 3 is used on the P90 (σ denotes packet size):

$$\lambda \approx \begin{cases} 1 & \text{if } \sigma \geq 4096 \\ -\frac{1}{1878}\sigma + 3.238 & 1024 \leq \sigma < 4096 \\ \frac{2201}{\sigma} & 0 < \sigma < 1024 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Total CPU time $\mathcal{U}$ needed for L^4 ATM and ATM driver processing of n transmit flows and m receive flows is thus computed as follows:

$$\mathcal{U} = \frac{\sum_{i=1}^{n} t_i \lambda_i}{t_{max}} + \frac{\sum_{j=1}^{m} r_j \lambda_j}{r_{max}} \quad (4)$$

In Equation 4, t_i indicates a guaranteed transmit flow, r_i a guaranteed receive flow, t_{max} maximum achievable transmit bandwidth and r_{max} maximum achievable receive bandwidth. $\mathcal{U} = 1$ indicates full CPU utilization.

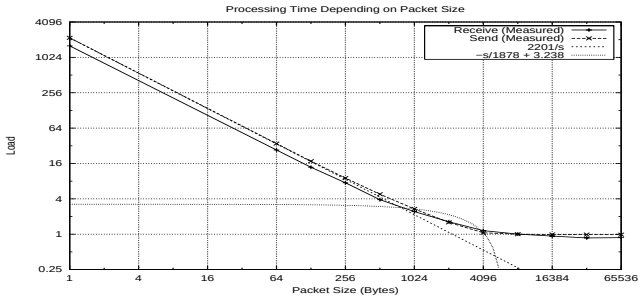

Figure 6. CPU load for a 40 Mbps transmit connection (P90).

Enforcement of CPU reservation is not yet integrated in the current prototype, but implementation of DROPS' scheduling framework and system-global CPU resource management is underway. CPU time reservation is emulated using priority hierarchies in conjunction with the policing function.

A temporary solution is used for the problem of incoming traffic overloading the receiving machine. A high-priority driver will consume all available CPU time, transmit incoming data to the protocol's *pseudo interrupt thread* which will buffer the data for readers. However, readers will get preempted almost instantly by the driver, leading to excessive data loss due to overflowing buffers in the protocol. A workaround, using a low priority for the device driver, has proved satisfactory in the implementation. Obviously, the problem will be permanently solved by integrated scheduling support, driver real-time support and appropriate configuration of the network.

[5] 256 KB Cache, 64 MByte RAM, Intel Neptune-based boards

[6] 256 KB Cache, 64 MByte RAM, ASUS main boards P/I-XP6NP5

[7] That is, CPU time required for a 80 Mbps flow is assumed to be twice as high as for a 40 Mbps connection.

Management of network bandwidth and core buffers are handled straightforward. L^{4}ATM is configured with the maximum achievable network bandwidth and mainly keeps track of the bandwidth used. Actual reservation is handled by the network either statically or dynamically. Buffer space is computed based on application-provided traffic parameters. Allocation and deallocation of buffer space is performed by the system's resource manager.

3.2. Implementation of UPC and policing

For Usage Parameter Control, values for interval length and credits are chosen based on the delay variation tolerance requested by the client (see Section 2.5). For jitter sensitive applications, small intervals (equaling 2 PDUs) are chosen, for the default case the algorithm tries to fit the equivalent of 10 PDUs into an interval, while jitter insensitive applications use larger intervals (equaling 50 PDUs).

When performing UPC, the current interval is determined by reading the L4 kernel's real-time clock. Thread suspension is implemented using L4 IPC, specifying a relative timeout. Thus, the suspended thread is woken up by the L4 kernel at the beginning of the next interval, when it resumes execution. Using the L4 real-time clock implies a maximum resolution of 2 ms on our test hardware.

3.3. Building the data path

When setting up connection parameters via the `setsockopt()` interface, L^{4}ATM establishes distinct permanently mapped memory segments for the transmit and receive direction. This has the advantage of speed and faster IPC compared to temporary mappings. Furthermore, it does not suffer from alignment restrictions. The driver interface is copy-free for the transmit direction (see Section 2.2). For the receive path, the current implementation still features one extra copy compared to the design presented.

Transmit and receive buffers are implemented as ring lists, where only current index position and data size are exchanged between L^{4}ATM and client library.

4. Evaluation

L^{4}ATM has matured during numerous stress tests performed in the last few months. While a multi-threaded design is traditionally prone to sporadic synchronization errors (implementing synchronization primitives used up a significant share of total programming time), L^{4}ATM routinely handles a number of concurrent `read()` and `write()` connections under competing load when running overnight.

Real-time overhead. At run time, additional cost is introduced by the call to the UPC function for each packet. The overhead associated with this function has been measured: It takes slightly more than 300 CPU cycles to call the UPC function, read the μkernel's real-time clock, check the conformity of the current packet, adjust connection credits and return to the caller. Cycles required on the P90 for the *complete* send path are in the range of $11577 \ldots 351060$, depending on packet size. No visible run time overhead is expected by the real time extensions. This has been verified experimentally.

Throughput. Using 8 KB sized packets, Table 1 compares throughput for L^{4}ATM against native ATM performance under Linux on the same hardware (P90).

		Avg	*Max*	*Min*
L^{4}ATM	**RX**	79.8	84.2	74.2
	TX	130.6	130.6	130.6
Linux	**RX**	58.3	62.8	52.4
	TX	129.5	130.6	126.4

Table 1. Throughput for 8 KB packets for Linux and L^{4}ATM.

Performance of L^{4}ATM was very satisfactory. Send performance is close to the maximum AAL5 rate. Average receive performance of L^{4}ATM was found to be 36.8 % higher compared with Linux. L^{4}ATM's gain is due to the smarter (e.g., more efficient) distribution of processing time: In Linux, data is enqueued in socket specific buffers by an interrupt service routine. A problem occurs when incoming ATM load is high. For the remaining path within the Linux kernel and within the application available processing time is not sufficient. Thus, a number of packets is dropped by Linux' interrupt thread due to the unavailability of buffer space at the socket queue. In contrast, L^{4}ATM makes sure that enough processing time is available for the *worker thread*. (Applications get guaranteed processing time by means of either DROPS' scheduling framework or static priorities.) This advantage outweighs the additional IPC operations required compared to Linux' in-kernel driver.

4.1. Predictability

Regarding deterministic behavior, Table 2 presents the requested sustainable data rate versus perceived data rates for two concurrent writer applications. After requesting a rate, both applications sent with full speed (In all scenarios in this section, applications are "dumb"—sending data in tight loops without QoS awareness.). L^{4}ATM transparently performs adaptation to the requested rate. The first two columns show the rate as requested by the application;

the third column the maximum data unit size specified by the application; the forth and fifth column displays interval length and credits chosen for UPC; the sixth column shows the data rate as measured by the writing application and the last column shows the received data rate on the peer machine.

Requested rate (Bytes)	Requested rate (Mbps)	Data size (Bytes)	Interval length (ms)	Credits (Bytes)	Rate sent	Peer rate
$8 * 10^6$	61.03	8192	10	79990	60.98	60.89
$3 * 10^6$	22.89	4096	13	39000	22.91	22.90

Table 2. Precision of UPC and policing algorithm for concurrent real-time `write()`.

Pre-scheduled writers. The previous measurement has sparked interest in the following question: How does L^{4}ATM's predictability depend on number and aggregated load imposed by several competing clients? To answer this question, a schedule has been pre-configured which tries to maximize the number of different combinations of concurrently running clients. During the total duration of about 4 hours, competing connections are periodically set up and terminated. Figure 7 shows the schedule used and gives information on the individual connections.

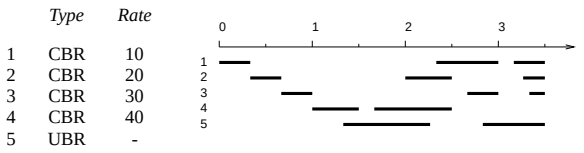

Figure 7. Concurrent writer schedule.

Results are given in Figure 8. Dots denote the received data rate as measured on the peer machine.

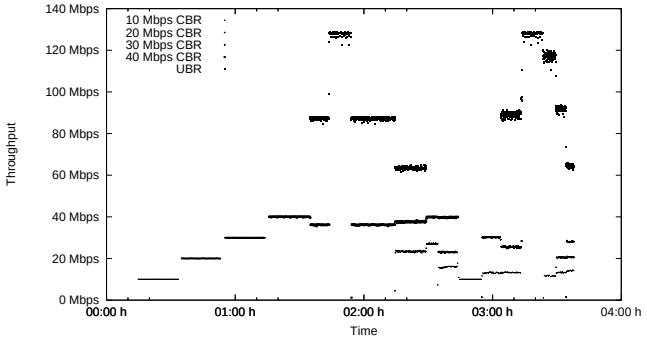

Figure 8. Five pre-scheduled concurrent writers.

During the first 80 minutes, the real-time connections ran separately. As expected from previous measurements, L^{4}ATM enforces the requested bandwidth perfectly. At the later stages of the measurement, competition between the connections exists.

The upper part of the plot displays the remaining bandwidth used by the UBR connection. On start and termination of real-time connections, L^{4}ATM's admission control decreases or increases the rate available for all best effort connections.

4.2. Separation—AIM

The commercial AIM multiuser benchmark suite VII [1] simulates typical multi-user application load. Using a set of subtests, relevant components of an operating system are stressed. The load mix used by AIM is designed to resemble typical system load. While increasing system load until response time becomes unacceptable (*cross-over*), the number of jobs handled by the system is measured[8].

A P90 machine ran both the time-sharing L^{4}Linux server and the L^{4}ATM server concurrently on top of the L4 μkernel. A fast peer machine (Pentium Pro 200 machine running Linux) was connected via a standard OC-3 multimode fiber and a FORE ASX-200WG ATM switch.

The test duration for the complete AIM measurement (until occurrence of cross-over) is about 10 hours. A L4 real-time application requested a guaranteed ATM connection with a constant bandwidth of 40 Mbps with 8 KB sized datagrams. Within a total time frame of 22 hours, about 5000 spot checks on received throughput on the peer machine—each consisting of 10000 datagrams or 80 MB data—were made.

Performance degradation of time-sharing component. Figure 9 compares the performance of L^{4}Linux in terms of jobs processed per minute.

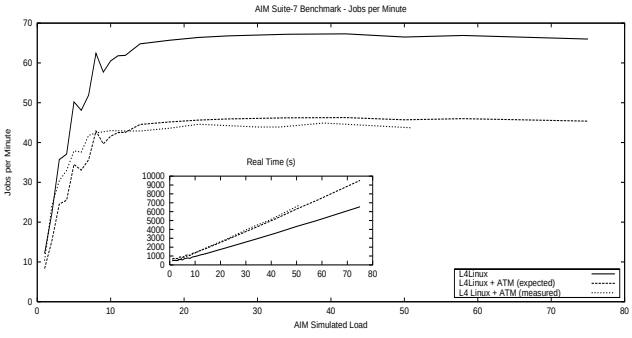

Figure 9. Achieved performance of the concurrently running time-sharing OS server.

In the top-most curve no real-time jobs exist—L^{4}Linux uses all hardware resources fully. Cross-over occurs at a

[8]The measurements are not certified by AIM Technology.

load index of 75. Under concurrent real-time ATM load (see lower curve), system performance was expected to suffer by about $31.2\,\%$[9]. Both estimated and measured performance are given. As can be seen from the graph, time-sharing system performance measured closely fits the estimate.

ATM performance under heavy local load. The second question of interest concerns L^{4}ATM's ability to preserve guarantees under heavy time-sharing competition. Figure 10 shows the bandwidth received on a peer machine. During measurement of samples $1 \ldots 2398$ AIM load increased until crossover occurred. At sample 2399, the AIM benchmark was completed. Except for periodic maintenance services in L^{4}Linux, only protocol processing was running on the test machine at this stage.

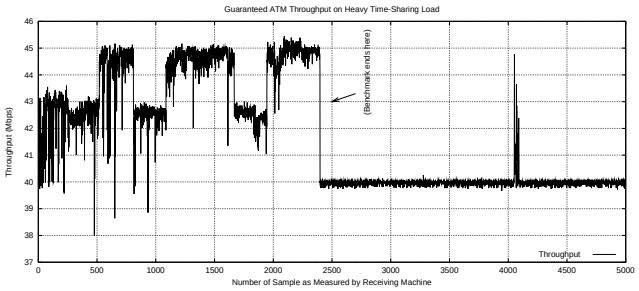

Figure 10. Measured throughput for guaranteed ATM connection under heavy concurrent load.

As can be seen from the results, L^{4}ATM always meets or exceeds the reserved rate, even under high load. However, the measurement also reveals an anomaly: While L^{4}ATM exactly meets the reserved bandwidth when using the machine exclusively, it exceeds the reserved bandwidth under extreme load conditions. This unexpected behavior is caused by the L4 μkernel, which wakes up suspended threads under this circumstances prematurely.

5. Related work

Similar to DROPS, Rialto [14] envisions coexisting real-time applications and time-sharing applications. Both pragmatic and practical assumptions have been made to aid actual implementation.

A theoretical model for QoS management has been proposed by [22] which intends to improve resource utilization for a single resource for perceived quality by the user. DROPS focuses on determining resource requirements, enforcing reservations and protection against time-sharing tasks. We do not explicitly optimize resource utilization, rather we provide techniques for integration of component-specific parameters sets into a generalized model.

Issues in designing zero-copy user-level protocols are subject of, for example, the U-NET project [25]. An experimental setup for a zero-copy architecture using ATM hardware is described in [15].

In the context of Real-Time Mach [16] many related aspects were subject to research. Predictable protocol processing has been attempted by separating protocol code from the Mach UNIX server [20] and linking it to applications (instead of a dedicated task protecting protocol code). Contrary to L^{4}ATM, resource reservation issues exclusively apply to processor time. The problem of priority inversion is addressed by dynamic adaptation of thread priority (instead of per-connection threads). Resource requirements are estimated by monitoring (instead of measuring). Applications are expected to scale (instead of L^{4}ATM's assumption on "dumb" applications).

A related approach with very similar goals on designing end system's communication subsystem is proposed in [19], based on the *x*-kernel. A thread-per-connection model is favored there, too. L^{4}ATM improves on this approach in several ways: (1) Multitasking in L^{4}ATM is fully preemptive, and not cooperative. (2) Suspending threads in the L^{4}ATM implementation is work-preserving (e.g., no busy waiting is involved). (3) Guarantees in L^{4}ATM hold under existence of exclusively misbehaving time-sharing or real-time applications. (4) L^{4}ATM uses an off-the-shelf network adapter (as opposed to a software null device).

6. Conclusion and future work

The DROPS architecture, simultaneously supporting real-time and time-sharing applications, has been introduced. DROPS integrates multiple personalities, one such component—an ATM-based protocol server targeting deterministic network communication support—is the centerpiece of the work presented.

Design and implementation of L^{4}ATM has been discussed and selected aspects of the implementation on top of the L4 μkernel have been described.

Experiences with L^{4}ATM give us reason to be optimistic. Using highly implementable, efficient L4-based techniques for host resource management, UPC and policing, a running system has matured. Applications are provided with a standard interface, and do not need particular QoS awareness.

L^{4}ATM outperforms classic monolithic kernels in terms of throughput. Traffic management algorithms have been shown to work in real-world scenarios, enforcing precise control over requested rates.

Separation from time-sharing traffic has been demonstrated by running a benchmark on the time-sharing L^{4}Linux server. Overhead caused by implementation of

[9] Based on L^{4}ATM's resource requirements estimate.

real-time support is well below 1 %, except for very small packet sizes.

The mechanisms designed and implemented have been integrated into a prototype "DROPS chain" (a media server formed by a real-time file system and L^{4}ATM) and have passed initial tests. We expect to learn more from refining this scenario.

The main future task involves integration of DROPS' scheduling mechanisms. Experiments with staggering streams will lead to refinement of interval length determination. Porting Linux' ATM signalling support to native L4—to allow for generic propagation of traffic parameters to the network—is a useful implementation step.

7. Acknowledgements

We are indebted to the members of the DROPS group at TU Dresden for supporting design, implementation, evaluation, integration and presentation of L^{4}ATM. Among them, Michael Hohmuth, Jean Wolter, Sven Rudolph and Uwe Dannowski deserve special mention.

We also thank the anonymous reviewers for their comments, in particular for their helpful critical remarks.

Finally, the Linux-ATM effort by Werner Almesberger served as code base for the protocol.

References

[1] AIM Technology. *AIM Multiuser Benchmark Suite VII*.

[2] W. Almesberger. Linux ATM API (Version 0.4). Technical report, LRC Lausanne, 1996.

[3] W. Almesberger. ATM on Linux. http://lrcwww.epfl.ch/linux-atm/, 1997.

[4] D. P. Anderson. Metascheduling for Continuous Media. *ACM Transactions on Computer Systems*, 11(3):226–252, August 1993.

[5] M. Borriss and U. Dannowski. Linux Support for FORE Systems PCA-200E NIC. http://os.inf.tu-dresden.de/project/atm/, 1997.

[6] U. Dannowski. An ATM Driver for DROPS. Term paper (Großer Beleg)—Dresden University of Technology., 1998.

[7] P. Druschel and L. L. Peterson. Fbufs: A High Bandwidth Cross-Domain Transfer Facility. In *ACM Symposium On Operating System Principles (SOSP)*, 1993.

[8] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *ACM Symposium On Operating System Principles (SOSP)*, Saint Malo, France, December 1997.

[9] C.-J. Hamann. On the Quantitative Specification of Jitter Constrained Periodic Streams. In *MASCOTS*, Haifa, Israel, January 1997.

[10] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.

[11] H. Härtig, J. Wolter, and J. Liedtke. Flexible-Sized Page Objects. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 102–106, Seattle, WA, Oct. 1996.

[12] H. Härtig, R. Baumgartl, M.Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS - OS Support for Distributed Multimedia Applications. In *Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[13] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel Based Systems. In *ACM Symposium On Operating System Principles (SOSP)*, Saint Malo, France, December 1997.

[14] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. B. III. Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System. In *Network And Operating System Support for Audio and Video (NOSSDAV)*, 1995.

[15] H. Kitamura, K. Taniguchi, H. Sakamoto, and T. Nishida. A New OS Architecture for High Performance Communication over ATM networks. In *Network And Operating System Support for Audio and Video (NOSSDAV)*, 1995.

[16] T. Kitayama, T. Nakajima, and H. Tokuda. RT-IPC: An IPC extension for Real-Time Mach. In *Proceedings of the Winter 1992 USENIX Conference*, pages 91–104, Sept. 1993.

[17] J. Liedtke. On μ-Kernel Construction. In *ACM Symposium On Operating System Principles (SOSP)*, Copper Mountains, CO, December 1995.

[18] J. Liedtke, H. Härtig, and M. Hohmuth. OS Controlled Cache Predictability for Real-Time Systems. In *Third IEEE Real-time Technology and Applications Symposium (RTAS'97)*, Montreal, Canada, June 1997.

[19] A. Mehra, A. Indiresan, and K. G. Shin. Structuring Communication Software for Quality of Service Guarantees. In *17th IEEE Real-Time Systems Symposium (RTSS)*, December 1996.

[20] C. W. Mercer, J. Zelenka, and R. Rajkumar. On Predictable Operating System Protocol Processing. Technical Report CMU-CS-94-165, School of Computer Science, Carnegie Mellon University, May 1994.

[21] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Operating System Design and Implementation*, 1997.

[22] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *IEEE Real-Time Systems Symposium*, December 1997.

[23] S. Sathaye, editor. *Traffic Management Specification Version 4.0*. ATM Forum Technical Committee, Traffic Management Working Group, 1996.

[24] G. Vattrodt. Entwicklung einer ATM-basierten Echtzeitkomponente auf dem Mikrokern L4 (in German). Master's thesis, Dresden University of Technology, 1997.

[25] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *ACM Symposium On Operating System Principles (SOSP)*, Copper Mountains, CO, December 1995.