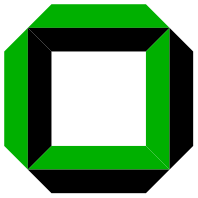

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme

Lehrstuhl Systemarchitektur

Asynchronous Communication Using Synchronous IPC Primitives

Stefan Götz

Diplomarbeit

Verantwortlicher Betreuer: Prof. Dr. Alfred Schmitt
Betreuende Mitarbeiter: Dipl.-Inf. Volkmar Uhlig
Dipl.-Math. Gerd Liefländer

30. Mai 2003

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 30. Mai 2003

_____

Stefan Götz

Abstract

The asynchronous communication model provides applications with parallelism, message buffering, and a convenient programming model. In contrast to the synchronous model however, it is inherently associated with policy regarding the management and delivery of buffered messages. Communication partners are dependent on controlling these policies in order to achieve additional semantics and performance optimizations customized to their communication behavior.

Many existing operating systems implement asynchronous IPC primitives at kernel level but suffer from significant performance penalties compared to synchronous systems. Also, buffer management policies are hidden from the applications. Communication frameworks focusing on performance achieve high throughput even across multiple protection domains. However, they also imply memory management related policies and custom application trade-offs between throughput and latency are not possible.

This thesis describes how the advantages of both communication models can be combined by emulating asynchronous communication on top of synchronous IPC primitives. It discusses the emulation with regard to performance, trust and protection between the involved communication partners, and transparency towards existing protocols. The presented concepts are evaluated on the L4 micro kernel. It is shown that asynchronous communication is achievable at user level with performance comparable to the synchronous IPC it is based on while preserving protocol transparency and flexibility for application specific policies.

Acknowledgments

I would like to thank my supervisors Volkmar Uhlig and Gerd Liefländer. Their expertise, patient support, and encouragement made this work possible.

The members of the System Architecture group in Karlsruhe have been a constant source of ideas and suggestions. In particular, I am thankful for the insightful discussions with Andreas Haeberlen and Uwe Dannowski's thorough proof reading.

Contents

Chapter 1

Introduction

Communication across protection domains is fundamental to securely modularize software systems. Modularized designs can benefit from reduced complexity, higher flexibility, and enhanced customizability. In order to achieve safe interaction of untrusted components and fault isolation, components can be placed into their own protection domains. This is of particular importance for overall robustness of component based operating systems.

Separating components by protection boundaries incurs a performance overhead on the interaction and communication between those components. However, components need to interact frequently. For example, non-trivial service requests often need to be handled by a series of interacting components with specific functionality instead of a single monolithic entity. Thus, the performance impact of cross-domain interaction is multiplied. Especially applications with high-bandwidth demands such as file and web servers suffer from the costs of multiple cross-domain data transfers and have received attention from researchers [5, 13] on classic and early micro kernel systems.

Modern micro kernels are highly suited for modular system designs. They provide simple and well-understood abstractions for the fundamental building blocks of a system: execution contexts and protection domains. This minimality is a primary design principle of the L4 micro kernel resulting in a flexible and policy-free interface and very good performance of its primitives, in particular for inter-process communication (IPC).

The performance of the IPC primitive of L4 has in part to be attributed to its simple synchronous nature, i.e. communication is blocking and unbuffered. An important alternative model is asynchronous buffered communication. Asynchrony is necessary to achieve parallelism by overlapping communication and computation and thus, e.g. to increase CPU utilization. Server components for example can exploit parallelism in order to continue providing service to their clients while communication with other components is in progress.

A synchronous message transfer across a protection boundary causes a

switch between the protection domains in most cases. This implies execution time overhead as the address spaces need to be switched which requires kernel involvement. Also the consequential costs due to cache flushing can be significant on architectures such as IA-32. Asynchronous IPC can help to avoid this problem because messages are buffered, thus the number of address space switches can be reduced. Together with other optimizations, these properties of asynchronous communication allow to create very efficient facilities for cross-domain message transfer. Consequently, performance close to monolithic systems becomes possible while maintaining the benefits of modularization.

However, asynchronous communication is also inherently associated with the additional costs of handling buffered messages, in particular the overhead of copying messages into a temporary buffer. This reduces the effectiveness of the hardware caches and incurs consequential costs on applications.

Furthermore, the buffer management is affected by a large number of parameters, e.g. the location of the buffers, resource accounting, or real-time guarantees, which are of different importance to specific applications. It is very difficult to expose all ranges of policies inherent to asynchronous IPC to the user without dynamic kernel extensions.

The synchronous and asynchronous models can be emulated by each other. Thus, asynchronous semantics can be achieved on top of synchronous primitives without modifying the kernel. At the same time, all aspects related to semantics, policies, and performance can be addressed at user level and adapted to specific application demands.

This thesis shows how to achieve asynchronous communication semantics based on synchronous IPC primitives. Its fundamental approach is to obtain parallelism and thus asynchrony by delegating the execution of synchronous primitives to other threads. It explores optimization techniques to enhance performance for different application demands characterizing the results with regard to protection, transparency, and performance with a focus on the IA-32 architecture. The applicability of the resulting designs ranges from asynchronous communication that is fully transparent to synchronous communication partners to high-bandwidth communication across multiple domains as is common in multi-server I/O scenarios.

Chapter 2

Background

This chapter gives an overview of the L4 micro kernel. It reviews the costs relevant for IPC with focus on the IA-32 architecture based on the extremely fast L4 IPC primitive. It analyzes the inherent properties of asynchronous communication compared to synchronous communication and provides an introduction to the structure of multi-server operating systems and their particular requirements on communication.

2.1 L4

The L4 micro kernel was originally developed by Jochen Liedtke. Several designs and versions have emerged from the original ideas. In this document we will relate to the Version 4 API [15] developed and implemented at the University of Karlsruhe.

L4 is based on threads and address spaces as its fundamental abstractions. For communication between threads, L4 offers synchronous, i.e. unbuffered, inter-process communication (IPC). Other aspects of its functionality, such as interrupt or page fault handling, are mapped to these abstractions and mechanisms. Classic OS services like paging and hardware device handling are completely exported to user level.

Address spaces implement virtual memory and serve as protection domains. They are recursively constructed by user-level pagers. An initial address space, *Sigma0*, contains all physical memory in an idempotent mapping. Further address spaces are populated by creating mappings of virtual memory regions between these spaces.

While mapping establishes shared memory leaving the memory accessible to the sender the grant operation has move semantics so that only the receiver can access the granted memory. Both operations are integrated with the IPC primitive requiring consent of the partner to receive a mapping. The receiver implicitly accepts that the mapping can be revoked at any time.

L4 exports paging to user level by establishing a user-configurable association of every thread with a *pager* thread. On a page fault, the kernel synthesizes a blocking IPC from the faulting thread to its pager. The pager resolves the fault by establishing a memory mapping and replying to the faulting thread. On the reply, the kernel unblocks the faulting thread.

Threads have globally unique identifiers which are managed by a privileged thread. It is also responsible for creating and destroying threads. The in-kernel scheduler dispatches threads based on a fixed-priority round-robin policy.

With the IPC primitive of L4, threads can communicate in a synchronous fashion. Since messages are not buffered by the kernel, an IPC is only successful when sent to a thread that is ready to receive. I.e. a rendezvous has to occur between the sender and the receiver. The time spent blocking on a busy sender or receiver can be limited via timeouts. Receivers may choose between accepting messages from a specific (*closed wait*) or any thread (*open wait*). IPC interacts with scheduling in that a send operation causes a thread switch to the receiver only if it has a higher priority than the sender.

An IPC message descriptor is stored in a static buffer private to each thread. It contains both data and meta data. The latter either references a memory area containing additional data or descriptors for address space operations. A similar structure allows receivers to specify where incoming data or mapped memory is to be placed by the kernel.

The propagation feature enables senders to impersonate other threads. This is transparent but detectable by the receiver who also learns the identity of the true sender. Propagation is permitted if the true sender resides in the faked sender's or the receiver's address space. A privileged thread can control propagation between disjoint address spaces.

All aspects of L4 design - architecture, algorithms, interface, and implementation - focus on minimality in order to achieve a maximum of performance. [9] reports that only the synergy of multiple optimization techniques on all design levels results in such low costs. Special attention is paid to hardware caches and how crucially performance depends on caching effects. An important property of L4 is its small impact on caches, in particular for short IPC system calls. Thus the performance improvements of caching remain with the applications despite the communication via the kernel. This aspect is of even greater importance today as the gap between processor and memory speed has steadily grown and continues to do so. Thus, caching becomes more and more crucial for performance.

2.1.1 IPC Performance

Three important factors in IPC performance can be deduced from Liedtke's observations:

Data copying: the process of copying data itself is bound by memory bandwidth which is slow compared to today's CPU speeds. Since the hardware caches are of limited size, they become ineffective when copying large amounts of data and the cache pollution caused by copying incurs consequential costs.

Context switches: the IA-32 architecture implements un-tagged translation look-aside buffers. Thus, it is necessary to flush the TLB when switching from one address space to another causing consequential costs for re-populating the TLB. Also virtual caches, e.g. the trace cache of the Intel Pentium 4, need to be flushed and re-populated on context switches.

Entering and leaving the kernel: the base costs of system calls have led Gefflaut at al. [7] to the observation that despite L4's well performing IPC primitive communication costs need to be reduced by avoiding system calls where possible.

Although these factors are platform dependent, kernel involvement is required for IPC in order to securely cross protection domains. The kernel needs to install the new protection domain and possibly change the execution context. Thus, synchronous message based communication across protection domains is associated with certain costs which can not be circumvented. Bershad et al. [3] proposed a user-level IPC facility to cross protection domains but it is restricted to shared memory multi-processor systems and user-level threads.

2.2 Implications of Asynchronous IPC

Asynchronous IPC is inherently associated with implications on the performance and the policy of a message transfer. This is one reason for synchronous IPC in L4.

In order to implement asynchronous communication, all messages that have been sent but not delivered need to be stored in an intermediate buffer until the receiver consumes them. This is to achieve copy semantics so modifications of the buffers of a sent message do not affect the message contents. In contrast to synchronous IPC, where the message can be copied directly from the sender to the receiver, messages need to be copied twice for asynchronous IPC. Thus, the asynchronous IPC operations inherently increase cache footprint.

Copy semantics for asynchronous page mappings pose a similar problem. Although in this case the process of copying can be deferred using the copy-on-write technique, copying is still required when the mapped pages are modified by the sender or the receiver. Consequently, L4 offers only share or move semantics for memory mappings.

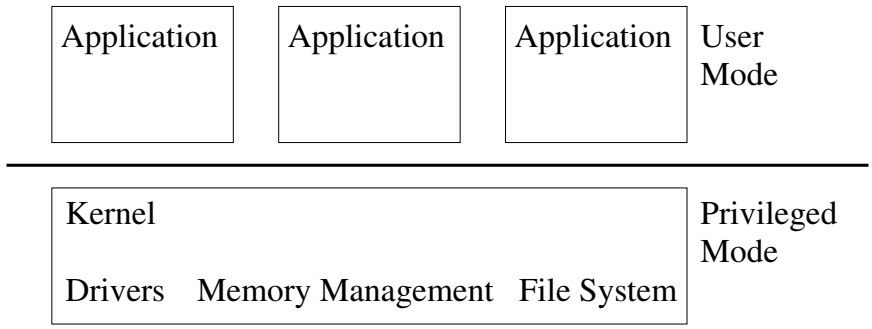

Figure 2.1: Structure of a monolithic operating system

A number of parameters can control the asynchronous communication process, such as the order in which buffered messages are delivered to a receiver, where they are stored in memory (which can be of importance, e.g. for cache-coloring), real-time guarantees, and how the message transfer facility accounts for the memory holding the buffered messages. Furthermore, new variables are introduced by new application demands. Thus, it is hard, if not impossible, to export all policy associated with asynchrony to the user.

Since asynchronous communication can be emulated by synchronous communication and vice versa [6], it is not necessary to implement both models in the micro kernel. To do so would mean to violate the minimality principle of L4 which is a key to its performance. Thus, it is reasonable to realize a fast and flexible synchronous IPC primitive in the kernel and emulate the asynchronous model at user level giving access to all policy and performance trade-offs as applications demand.

With this approach the synchronous primitive forms the performance base-line for asynchronous communication. Additional costs arise from the emulation of asynchrony.

2.3 Communication in Multi-Server Systems

Traditional monolithic operating system implement a large set of services. For example, drivers for device access, memory management, user interfaces, and the necessary infrastructure for interaction between subsystems (see Figure 2.1). The operating system kernel executes in a privileged processor mode while applications execute in a non-privileged mode. Thus, the kernel can be protected from the applications. At the same time, all subsystems of the kernel have to be trusted not to compromise the rest of the system.

The monolithic approach not only emerged naturally from simpler systems, it also allows to perform any kind of optimization across subsystems. However, monolithic systems have already grown to a very high level of

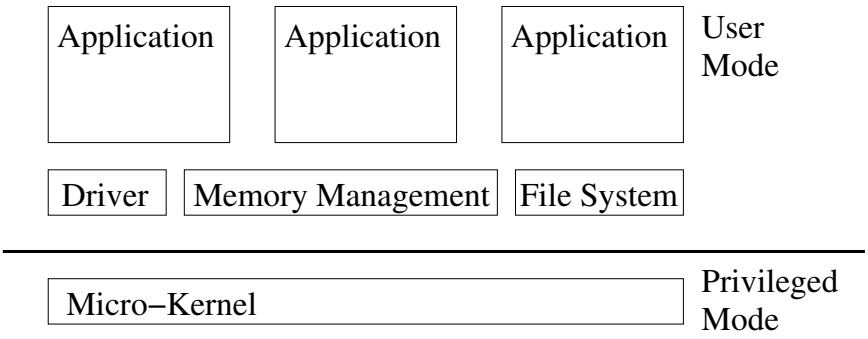

Figure 2.2: Structure of a multi-server operating system

complexity and continue to do so with every additional feature required by new application demands. Thus, it becomes increasingly hard to maintain them and to ensure security and reliability.

Multi-server operating systems take a different approach. System services are separated into modules executing at user level in their own protection domains, which are typically represented by virtual address spaces (see Figure 2.2). A micro kernel provides protection and allows modules to safely communicate across address space boundaries. From the kernel's perspective there is no difference between modules implementing system services and regular user applications accessing those services. Instead, the protection mechanisms offered by the kernel allow to safely manage the permissions to perform otherwise privileged operations, e.g. device access or page table manipulations.

The complexity of operating systems is addressed by multi-server systems using the standard software engineering technique of modularization. Subsystems are isolated and their interactions and interdependencies are restricted to well-defined interfaces. Extending the system functionality is usually localized to a small number of modules. Similarly, existing modules can be added or removed from the system as required by the environment or as performance needs dictate.

When a subsystem is protected except for the interface it exports via IPC, there are few possibilities to compromise it. Thus, it becomes less likely that a flawed module can affect other parts of the system. At the same time it becomes possible to safely extend the system with untrusted modules.

A service can require the interaction of several modules, e.g. a file access could involve the file system and the disk driver. Significantly deeper nesting than in this simple example is not unlikely in larger systems. Since protection is enforced by the kernel, communication across address space boundaries requires kernel involvement. This makes subsystem interaction

more expensive compared to a monolithic system where all data is shared and directly accessible.

2.3.1 Message Passing

Message passing IPC primitives allow to easily emulate common programming models. The remote procedure call (RPC) paradigm extends the notion of a local function call to a service invocation, e.g. across protection or machine boundaries. Thus, a caller sends a message containing the arguments to the callee and blocks. The callee identifies the operation to be performed on behalf of the caller as part of the communication protocol. After the operation finished, the callee re-activates the caller by sending a reply message which contains the results of the operation.

The similarity to local function calls makes RPC simple and integrate well with existing programming models. Tools exist which automatically generate the communication code.

Message passing can also be used to model the transfer of messages as parts of a larger data stream. Splitting data into multiple messages is sometimes necessary due to protocol requirements or restrictions of the communication primitives, such as a maximum message size. Grouping fixed or variable amounts of data into message is common in communication and particularly network protocols e.g. to temporarily store and then forward the data or for fault tolerance. Consequently, a message serves as a partial conversion of time-multiplexed data to a space-multiplexed form.

Messages and thus data travel from a source to a sink. They are often processed by several software layers with distinct functionality. In multi-server systems, these layers are often represented by components which reside in their own protection domains. Any layer in a system can act as source and sink and thus also forward messages.

We will call the sequence in which data traverses software layers the *data path*. This path does not only depend on static factors such as the data contents and the interaction of the involved layers but also on dynamic aspects, e.g. user configuration and load balancing. Furthermore, data paths from an incoming layer can diverge to several outgoing layers and vice versa. Thus, the complete path that data is to take through a system can not always be determined by the initial source.

A data path provides locality, i.e. often more data can be expected on the same path within a certain time frame [12]. An obvious example is the transfer of a large file where all data will usually travel on the same path from the disk driver through the protocol stack to the network interface driver. In protocol design, the locality of data paths can be exploited for performance optimizations.

2.3.2 Message Buffers

Messages can be represented by buffers containing a fixed amount of the data in contiguous virtual memory. After a buffer is allocated, this simple representation is sufficient for read only access to the message contents inplace modifications.

More complex operations are common-place in network protocols storing messages in buffers. They add or remove protocol information to the beginning or the end of a buffer. Also the contents of multiple buffers are joined to form larger messages or a buffer is split into smaller pieces. Since a buffer can not be shrunk or grown in size (unless additional space is pre-allocated in each buffer), such operations have to be realized by allocating additional buffers and copying the relevant data. This causes increasing overhead with larger message sizes.

An abstract data type (ADT) can help to avoid data copying when modifying a buffer. It introduces a level of indirection describing an aggregate of buffers referring to parts of their contents. The join and split operations are reduced to logical operations on the ADT. For adding data to the message, additional buffers need to be allocated and added to the aggregate. BSD mbufs [11] are an example of buffer aggregation based on an ADT.

Furthermore, the contents of buffer aggregates can be modified without changing the contents of the buffers. Instead, new buffers are allocated, the modified data is written to them, and they are inserted into the aggregate where the modifications are to appear, leaving the original buffers unchanged.

2.3.3 Transfer Semantics

When passing a buffer between protection domains, the involved domains can have different sets of access permissions to the buffer after the transfer is complete, which we call the transfer semantics.

Copy Both sender and receiver domain retain access to a private copy of the buffer. Thus, subsequent modifications are only visible locally. The copy-on-write optimization delays the copying until one of the buffers is modified. However, the overhead introduced by copying is unacceptable for large amounts of data.

Share After a buffer has been transferred, the sender and receiver domain have access to a single copy of the buffer and modifications of the buffer are immediately visible in both domains. The same holds for buffers transitively shared among multiple domains. Sharing can often be combined with modifying access rights of the buffer, e.g. the sender can pass only read rights to the receiver but retain read and write permissions.

Move The sender loses access to a buffer when passing it into a receiver domain. Similar to copy-on-write, move can be performed lazily so that the receiver can revoke the access rights of the sender for an initially shared buffer.

2.3.4 Transfer Mechanisms

Many IPC implementations provide copy semantics to enforce protection via private message buffers. But they suffer from the overhead of explicitely copying the data. Druschel and Peterson evaluate the characteristics of two other mechanisms commonly available: page remapping and static shared memory [5].

They conclude that sharing memory statically can not safely eliminate all copying. A globally shared memory area accessible by all protection domains violates data privacy. Pair-wise shared memory requires copying when data is to be forwarded into a third domain and group-wise shared memory requires that a source domain can determine the complete data path of a buffer before its allocation.

Instead, virtual memory pages containing buffers have to be passed dynamically between protection domains. Virtual memory systems support different semantics of such transfers: move in System V, copy(-on-write) in Mach, share and move at kernel level and copy at user level in L4.

2.3.5 User-Level Paging

Modern micro kernels, such as L4 [10] and Eros [14], export virtual memory management to the user. This is safe because the kernel maintains protection boundaries and ensures that a user task can fully control the manipulation of its address space.

The basis for user-level paging are memory mappings and page fault handling by the user. When a virtual memory region is mapped from a source to a destination address space, the kernel establishes page table entries so that the region in the destination space refers to the same physical memory as in the source address space. A mapping can have copy, share, or move semantics depending on the virtual memory system implemented by the kernel.

User-level page fault handling requires the kernel to reflect page faults to a handler provided by the user. This handler can then resolve the page fault by mapping memory to the faulting thread or by requesting a mapping to the faulting thread from another entity.

With user-level paging, the page remapping and shared memory mechanisms are unified as user tasks are responsible to establish shared memory via memory mappings. Also each memory buffer is associated with a memory provider. The memory provider is responsible for mapping the memory

that backs the buffer to the address space in which the buffer is to be accessed. The fact that also the sender and the receiver of a buffer can act as its memory provider offers a large design space with regard to overall system structure, performance, and resource management.

Chapter 3

Related Work

This section gives an overview on earlier work on asynchronous communication models.

3.1 Mach

The Mach micro kernel [1] implements an asynchronous IPC primitive. Liedtke provides a detailed performance comparison of L4 and Mach in [9]. It is shown that the performance of Mach IPC suffers from copying a message twice, as we discussed in section 2.2 . As a result, the costs of a message transfer increase about twice as fast compared to L4. Combined with the high base costs of a Mach IPC, this incurs a substantial overhead on communication.

Furthermore, Liedtke analyzes the cache behavior of Mach and its effects on system performance in [10]. He shows that Mach has a significant cache footprint. Thus, applications have to re-establish their cache working sets after system calls and the overall system performance is degraded.

3.2 Mbufs

The BSD operating system avoids the overhead of copying messages when data needs to be added to or removed from message buffers. These are commonplace operations in network protocols.

Messages are internally represented as mbufs [11]. They are abstract data types referencing and describing a data buffer. Modifying, concatenating, and stripping buffer contents thus become logical operations. They require to modify the meta data in the mbuf structure and potentially the allocation of further mbufs. However, the buffer contents remain unmodified.

3.3 Fbufs

With fbufs, Druschel and Peterson [5] acknowledge the importance of efficiently crossing multiple protection domains in multi-server systems. The fbuf approach circumvents the costs of copying data in cross-domain transfers by dynamically establishing shared memory via page re-mapping. Focusing on user-to-user throughput for high-bandwidth network connections, they employ a number of very effective optimizations in order to avoid all per-page and per-message costs for cross-domain transfers in the common case.

Message buffers are introduced as primordial objects into the messaging and memory management system. A centralized buffer management is responsible for establishing memory mappings so the communication partners can access and transfer the buffers.

Buffers are only writable for the allocating protection domain and only until they are sent to a different domain. Thus, the receiver domains can rely on the buffer not being modified. Buffer aggregates similar to mbufs are introduced which allow modifications at the aggregate level in the domains receiving the buffer.

Based on locality of communication (see section 2.3.1), memory mappings stay established after a buffer has been de-allocated. The buffer remains associated with the memory mapping and is put into a pool dedicated to the data path. New buffers for the same data path can be allocated directly from that pool, but write access needs to be restored to the allocating domain. It is necessary to determine the data path of a message before allocating a buffer for it. When all sending domains are trusted not to modify the buffer contents, write permissions can be retained in those domains, eliminating the need to toggle write access permissions.

Druschel and Peterson show that page re-mapping significantly reduces the costs of cross-domain communication for large amounts of data and even for small messages when all optimizations can be applied. The efficiency of the framework results from the presented optimizations which eliminate the per-page and per-buffer costs of the page remapping mechanism in the common case.

The fbufs approach relies on an IPC mechanism to indicate that a message was sent. The reference implementation benefits from the asynchronous nature of Mach IPC.

3.4 IO-Lite

IO-Lite [13] addresses the fact that fbufs do not support a unified I/O framework which can be used to transfer all I/O data efficiently across protection boundaries. IO-Lite uses the same basic concepts as fbufs and extends the

buffer management to disk I/O data and the file cache.

Although IO-Lite does unify disk and network I/O data, it can not be seen as a truly unified framework which allows to efficiently pass arbitrary memory contents across protection domains without copying. This is mainly due to the assumption of centralized memory management. Consequently, a special purpose allocator serves the buffer memory making it necessary to copy data into the buffer before it can be efficiently transferred as presented. This is an acceptable restriction for I/O data which often needs to be copied from a device into main memory or vice versa. It does however not apply to devices which map on-chip memory into main memory areas. Copying would also be required to transfer data that originally is not allocated in an I/O buffer and resides in a different memory region.

Chapter 4

Design

This chapter describes how asynchronous communication semantics are achieved based on synchronous IPC primitives. It identifies design parameters relevant in multi-server operating systems and presents how the design can be optimized with regard to these parameters.

4.1 Assumptions and Prerequisites

The work presented in this chapter is inspired by the abstractions and mechanisms available in the L4 micro kernel and implemented in the SawMill multi-server operating systems. However, the design is applicable to more abstract and thus other systems as well. This section lists the fundamental assumptions our design relies on.

Virtual Address Spaces: Virtual address spaces are expected to serve as protection domains enforced by the kernel. Thus, entities in separate address spaces can only interact with each other under the control of the kernel.
Physical memory is also represented by and accessed via virtual address spaces. The association between virtual and physical memory is managed by the user-level paging system.

Multiple Execution Contexts: The system has to support multiple execution contexts, i.e. threads. Each thread executes in an address space.

IPC: Message-based inter process communication is required which allows threads to communicate within and across address space boundaries. The communication is assumed to be unbuffered and blocking, i.e. synchronous. To ensure safe interaction between address spaces, the user needs to be able to control when communication takes place and how it affects its address space.

Paging: We assume a generic paging system for applicability to a wide range of systems, including user-level paging.

4.2 Parameters

Communication is a central aspect of operating systems and applications. Thus, the design of asynchronous communication as an additional model is affected by a large number of variables which determine whether it satisfies particular demands, such as its applicability to real-time or distributed systems. Based on the design assumptions presented in the previous section, we focus on three parameters which are important in the construction of a multi-server system:

Trust and Protection: Protection is a basic construction principle in multi-server operating systems. By placing components into separate protection domains, they can safely interact although they do not trust each other. All protection mechanisms have to be enforced by the underlying system which is implicitly trusted. With the given design assumptions, protection is provided by address spaces. Threads have to be able to safely interact via IPC across and within protection domains. Trust relationships between components also determine to what extent transparency and performance can be achieved.

Transparency: The interaction between subsystems or components follows certain protocols. For correct interaction these protocols must not be broken. In order to achieve transparency, the asynchronous communication must adhere to the given protocols. Communication in multi-server systems relies in most cases on IPC, thus the design addresses transparency with regard to protocols based on IPC.

Performance: The high degree of communication between components makes performance an important aspect. We concentrate on three key issues:

- Cache impact: the widening gap of processor and memory speed makes performance continuously more dependent on hardware caches. The performance of applications can suffer significantly from cache pollution, e.g. when data is copied in memory.
- System calls: architectures with long pipelines and out-of-order execution, such as IA-32, suffer from a high overhead of entering the processor's privileged mode and returning to user mode. Thus, system calls should be avoided when possible.

- Address space switches: on the IA-32 architecture, caches related to address spaces, such as TLBs and virtually tagged caches, need to be invalidated on address space switches. The overhead of repopulating caches in the new address space can be amortized by reducing the number of switches.

The design addresses the issues arising from these aspects and proposes adequate solutions.

4.3 Asynchronous IPC

The fundamental difference of the asynchronous communication model compared to the synchronous model is that messages are buffered. The buffering takes place when a message is sent. A communication partner may receive the message from the buffer at a later point in time. Thus, communication partners do not have to achieve a rendezvous situation, i.e. they do not have to ensure that one partner blocks until the other one also initiates communication.

The properties of an asynchronous communication system depend on the properties of the buffer, the buffer management, and the access protocol. The properties of a buffer are for example whether it is located in a separate protection domain or how many messages it can hold. The management of the buffer controls how messages are inserted or removed and how they are delivered. The access protocol determines how the buffer can be accessed externally, for example whether messages are exchanged via IPC or shared memory.

The design parameters affect these properties. To protect the buffer from the communication partners it can be placed into a separate address space. The buffer access protocol is restricted by transparency requirements. Performance improvements, on the other hand, often affect the protection and transparency parameters.

Buffers are a shared resource, as multiple communication partners can access them. This raises resource management issues. For example, a malicious thread can try to monopolize a buffer with a large amount of messages so that it can not be used by other threads. Access policies and protocols, e.g. per-thread quotas or maximum message life times, can be employed to prevent such situations.

4.4 Protocol Transparency

In order to transparently introduce asynchronous communication, existing protocols need to be followed. Within the design space a protocol is determined by the following properties:

Rendezvous Semantics: Synchronous protocols require rendezvous semantics. A message transfer is only successful if at a single point in time both partners have commenced the transfer. For most protocols and communication systems this implies that one of the partners has to block until the other partner performs the transfer. To match this property, the emulation of asynchronous communication needs to achieve rendezvous semantics towards the synchronous communication partners. Consequently, an additional entity is required, e.g. a placeholder or proxy, which blocks on a communication partner to handle a message transfer. Given the design assumptions, this entity is a thread.

With a synchronous protocol, a rendezvous point has to be provided for every message. For n incoming messages at a certain point in time, all of them are transferred only if there are n proxies ready to receive them. Similarly, with n pending outgoing messages, a rendezvous point for every message is only guaranteed to exist with n proxies blocked on sending each message.

Endpoints: Communication is performed between communication endpoints. They are associated with semantics, e.g. particular communication partners or a certain functionality. Thus, communication partners address each other or services via the names of endpoints. For transparency of asynchronous communication, the semantics and the names of communication endpoints have to be preserved.

Message Format: The format of a message determines its structure and semantics. For example, messages can be used to transfer data or to delegate rights. Transparency also requires the message format to remain unchanged.

Timing Constraints: A protocol determines how rendezvous semantics are achieved via timing constraints. For example, a server can require its clients to block for a minimum amount of time when sending a service request. Timing constraints often serve to detect misbehavior and to be able to react to it. Message sequences can also be subject to timing constraints, e.g. to limit latency or frequency of communication. Where such constraints are part of a protocol, they have to be preserved for full transparency.

4.5 Transparent Optimizations

As a starting point for designing a transparent emulation of asynchronous communication, a brute-force approach is taken. Message buffers and proxy threads were identified as fundamental requirements for achieving asynchrony. The message buffer, all proxy threads necessary to receive and

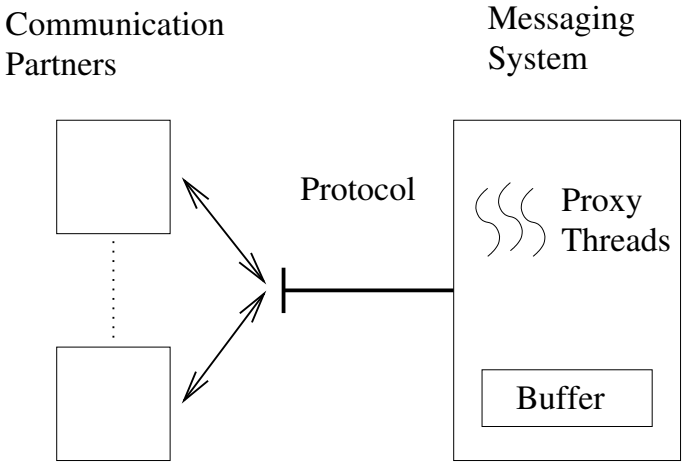

Figure 4.1: A transparent isolated messaging system

deliver messages, and the functionality to manage these entities are termed the *messaging system*. In order to protect it from malicious communication partners, it is placed inside a separate address space, as depicted in Figure 4.1.

The messaging system achieves transparency by implementing the protocol used between the communication partners. However, their communication becomes asynchronous since the messages are buffered by the messaging system. Thus, such a messaging system can be transparently interposed between synchronous communication partners to make them communicate asynchronously, if the timing constraints of their protocol allows to do so.

Since all partners and the messaging system itself reside in their own address spaces and communicate via IPC, protection is not reduced. Also the trust-relationships between the communication partners are not altered. However, the partners share the message buffer as a common resource. Thus, in order to detect misbehaving partners and to prevent denial-of-service attacks, the messaging system can put constraints on the communication protocol.

The presented approach can introduce significant overhead to a message transfer via the messaging system. For example, a large amount of proxy threads may need to be managed and additional copying is necessary per message. The following subsections address these issues.

4.5.1 Proxy Threads

As discussed in section 4.4, preserving rendezvous semantics requires a proxy thread per message. The costs of continuously creating and destroying threads can be alleviated by pooling and reusing existing threads.

Threads also cause costs by consuming resources. In particular, threads require kernel state, a stack, and potentially other resources at user level, such as meta data for thread management, thus increasing cache footprint. The consequential costs do not only affect the messaging system itself but also make it harder to achieve transparency, e.g. to meet timing constraints. Consequently, it is desirable to reduce the number of necessary proxy threads while maintaining rendezvous semantics.

The observation was made that one thread is required per message provided that no assumptions about scheduling behavior can be made. However, the underlying system may be able to guarantee, e.g. via strict priorities, that proxy threads are scheduled in favor of the communication partners. Then the communication partners can not initiate a message transfer as long as a proxy thread is active. A proxy thread becomes inactive by blocking for a message transfer. Thus, when a communication partner sends or receives a message, a proxy is guaranteed to be provided as a rendezvous partner. The messages are serialized on the proxy threads. Consequently, a single proxy can handle all incoming messages and it is sufficient to assign one proxy to each communication partner to which messages need to be sent.

Another optimization can be based on the timing constraints of the communication protocol. The protocol may ensure that the receiver waits for a message for a certain amount of time. Thus, the messaging system can delay a message for this amount of time without breaking the protocol. A proxy thread can exploit this fact to deliver messages to other receivers in the meantime. Consequently, the message delivery is serialized across receivers and less than one proxy per receiver is required.

The looseness of the protocol's timing constraints determines how many threads can be saved based on this optimization. When it is important to keep the costs caused by proxy threads low and transparency towards these timing constraints is not mandatory, the number of threads can be controlled via an additional protocol. This protocol limits the number of proxies based on, e.g. timing conditions or quotas per communication partner.

4.5.2 Co-Location

When the messaging system resides in its own protection domain, messages exchanged between communication partners have to cross an additional protection boundary. This additional address space switch is inherently associated with the costs in execution time due to the kernel and consequential costs, as virtually tagged caches, e.g. the TLB on the IA-32 architecture, need to be flushed and repopulated. However, these costs are unavoidable where the protection of address spaces is necessary.

Specific components or subsystems may rely on the messaging system to achieve asynchrony when interacting with their communication partners.

Thus, they directly depend on the messaging system to reliably and securely handle their communication requirements. As a result, their trust space extends to the messaging system. Provided that such components and the messaging system have the same set of privileges, they can be safely placed within a single trust domain and thus a single address space. This eliminates an address space switch per message which would otherwise be necessary.

The co-location optimization maintains protection and trust relationship between the communication partners. However, the common address space becomes a shared resource between the messaging system and the original component. Thus, the interaction of communication partners and the messaging system can also affect the original component. To prevent denial-of-service attacks exploiting this fact, resource management needs to be introduced.

4.5.3 In-Place Consumption

If messages are received into a static buffer, they can be overwritten before they are consumed. Thus, the messages need to be copied to another memory location before the next message is received. Copying the message in memory pollutes hardware caches and thus incurs costs on the messaging system.

To eliminate these costs, copying should be strictly avoided. Instead, each message needs to be received directly into a separate memory location. When messages are received, i.e. produced, faster than they are consumed, eventually all message buffers contain unconsumed messages. A protocol between the producers and the consumers of the messages can be established, which addresses this situation. For example, the producer may block until a buffer is consumed, the number of buffers could be increased, or message buffers could be overwritten after they exceeded a certain age without being consumed.

This optimization is only applicable if a receiver can specify to which location in its address space incoming messages are written.

4.5.4 Lazy Process Switching

To achieve asynchrony, messages are relayed via an additional IPC. Thus at least the overhead of one system call is added to an asynchronous message transfer.

Lazy process switching [8], introduced by Liedtke et al., safely exports kernel thread state to user level so that context switches within an address space can be performed without a mode switch. On L4 this mechanism is integrated with the existing thread and IPC abstractions. A user level context switch is performed as a *local IPC*. When an IPC between two threads blocks the sender and unblocks the receiver, the IPC can be executed

locally, i.e. without entering the kernel. The kernel state of the involved threads is updated on the next kernel entry.

Proxy threads can make use of lazy context switching when delivering buffered messages to local threads. In case the receiver is blocked waiting for the message, the system call is avoided and the context switch to the receiver thread can be performed at user level.

This optimization is only available on systems which provide lazy context switching and applies to co-located messaging systems. It is a pure performance optimization and does not affect trust, protection, or transparency.

4.5.5 Meta-Data Protocol

Message transfer via IPC causes cache pollution and execution time overhead. The proxy threads and the threads of the original component (the *worker threads*) exchange messages via IPC and thus suffer from these costs.

The copy and its costs can be avoided by changing the protocol between the proxies and the worker threads. Instead of transferring the complete message via IPC, only meta data is exchanged. In most cases it is possible to encode meta data in a significantly smaller form than the actual message, e.g. by using a pointer. Thus large messages can be exchanged with reduced overhead in execution time and smaller cache pollution.

4.6 Protocol Optimizations

The optimizations presented so far improved the internal interaction between the messaging system and a component, preserving transparency to the external protocol of the component. Since the external protocol is based on the IPC primitive, it implies certain costs due to the overhead associated with the system call, the address space switch, and the copying of messages.

When these inherent costs are unacceptable for the performance demands of particular applications, they can only be avoided by modifying the protocol. Thus, the transparency property is lost and the implementation of all involved communication partners needs to be adapted.

The original protocol is based on IPC for safe interaction of untrusting communication partners in separate address spaces. This is not necessarily the case when shared memory is established. Consequently, trust and protection issues that arise from protocol modifications need to be addressed.

This section discusses performance optimizations not transparent to protocols based on IPC between communication partners that potentially distrust each other. For each optimization, it describes the performance benefit, the optimization method with respect to trust and protection, and the protocol constraints under which it is applicable.

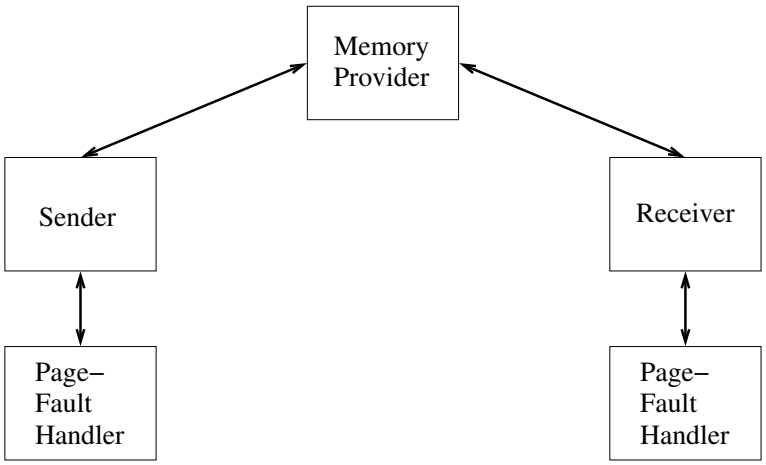

Figure 4.2: Shared memory with user-level paging

4.6.1 Shared Memory

The transfer of large messages between two address spaces via in-memory copying, as inherent to IPC, has two major impacts on performance. Namely, it consumes execution time and it pollutes hardware caches leading to consequential costs for the communicating and other applications in the system. Based on our design assumptions, shared memory is an alternative to IPC in order to cross protection domain boundaries. It avoids the direct costs of copying and reduces the cache footprint of a message transfer because the message is consumed in-place. Hereby, zero copy semantics can be achieved.

Figure 4.2 depicts the entities involved with a generic shared memory setup in systems with user-level paging. Memory is shared when two memory mappings exist from the same virtual memory range in the memory provider to both partners. The mappings can be established eagerly on request or lazily on page faults via the page fault handlers.

After two partners created a shared memory region, the message is transferred by the sender generating the message into that region and notifying the receiver with an IPC. Now the receiver can consume the message from the shared memory after receiving the IPC. The IPC serves two purposes: as with a copying message transfer, it acts as an activity transfer indicating that a message has been sent or received. Also, it can contain meta data describing the message, such as its location and size in the shared memory region.

Transparency is lost because shared memory needs to be set up and the message format can not be preserved. However, the message format can be customized. A sender may for example choose to pass only a part of the message to the receiver via shared memory and the rest of the message via

IPC.

When memory is shared between two address spaces, they are no longer strictly separated. Interaction is possible between them without kernel involvement. However, the kernel is assumed to ensure that the interaction is limited to memory modifications within defined regions in each address space. The following protection problems arise:

Mapping: it is not possible to access the shared memory if no memory mappings are established for it, e.g. due to a malicious memory provider.

Modification: the contents of messages can be modified while being accessed by the receiver. Thus, the time between the validation and the use of data can be exploited to invalidate it, known as the time of check vs. time of use (TOCTOU) problem. It arises if the semantics of a message are relevant for the receiver and it needs to interpret or validate the data. Otherwise, this is equivalent to the problem of the sender passing invalid or meaningless data to the receiver via IPC.

Resource Management: shared resources can be a source for denial-of-service attacks. Resource accounting and management protocols between the involved partners have to address this problem. Concurrent access can also be controlled via protocols.

If the partners trust each other and the memory provider, these protection problems are irrelevant.

In case the partners distrust each other but trust the memory provider, the memory can be assumed to be accessible within the timing guarantees given by the memory provider. If the receiver interprets the message contents, it has to validate them. Otherwise, the partner could supply it with invalid data. In this case, the TOCTOU problem arises. There are two measures for the receiver to protect itself:

Private copies: The receiver copies the parts of the message it needs to access from shared memory into private memory. Then it can check and access them without interference from the sender. However, the copying introduces overhead and reduces the performance benefit of the shared memory transfer.

Revoking access: The sender's write access to the shared memory is revoked after it transferred a message. It needs to be restored when the receiver consumed the message so the sender can reuse the buffer. These operations incur overhead and reduce the performance improvement by using shared memory.

An untrusted memory provider raises mapping and modification issues. Modification can be addressed as above. If no memory mappings are supplied by the memory provider, the partners need to be able to recover from

unhandled page faults. Depending on the underlying system, specific protocols have to be employed in such a situation. For example, the memory management or a timeout mechanism could be used for recovery. The partners have to ensure that they access the shared memory in a state that does not prevent the page fault handling or recovery. If the memory provider is untrusted but safety is a requirement, the system has to provide adequate recovery protocols.

To protect against untrusted partners, shared resources have to be managed based on protocols. They allow the involved entities to detect misbehavior and to react to it.

Protection is necessary against an untrusted partner that does not release shared memory after receiving a message. An adequate protocol is to limit the time the receiver may hold the shared memory buffer without releasing it to the sender. If the receiver does not release the buffer in time, it is misbehaving. The sender can react by aborting the communication with the receiver or by writing the next message into the buffer. If transparency towards external timing constraints is required, this resource management protocol has to be adapted accordingly or may not be applicable.

Depending on the format of the meta data, it can be used by the sender to compromise the receiver. If the sender is not trusted and able to specify meta data which describes memory not within the shared memory region in the receiver's address space, the receiver has to validate the meta data for correctness before using it.

Address translations necessary to access messages in the shared memory buffer consume execution time. They can be eliminated by placing the shared memory regions at the same virtual addresses in the address spaces of both partners. Given that the protocol allows it, e.g. information about the message format can be treated as implicit by both partners. Thus, it does not need to be transferred as explicit meta data thereby reducing the cache footprint of the IPC.

These optimizations are limited to messages which can be represented in memory. They are not suited for messages that e.g. transport access rights to communication endpoints or memory. Such messages have to be transferred via IPC.

4.6.2 IPC Coalescing

In the original protocol, each message is bound to an IPC, i.e. the activity is related to the message transfer. This implies that every message causes at least one system call and one address space switch. Additional IPCs might be required e.g. for resource management.

By transferring multiple messages per IPC, i.e. by coalescing the IPCs, these costs can be reduced. For IPC coalescing, the sender places multiple messages into the shared memory region. The IPC it sends to the receiver

then not only contains the meta data for a single message but for all messages in the buffer. Similarly, the receiver derives the layout of the buffer and the format of each message from the meta data. The same optimization can be applied to IPC based protocols that are used to indicate when a message is consumed by the receiver.

With this optimization, the receiver is not immediately notified of the availability of a new message. Thus, the latency of the message transfer is increased. If messages are produced at high rates, the costs of IPCs and context switches may limit throughput. In this case, the optimization provides additional throughput at the cost of increased latency. With an additional protocol between the communication partners, they can control this trade-off, e.g. by limiting the time the notification IPC may be delayed.

The increased latency through coalescing consumption notifications provokes shared resource management problems. It results in parts of the buffer being consumed but not immediately available to the sender. A larger buffer or a protocol can be employed to handle these issues. In a very simple protocol, the receiver waits for the sender to produce messages and send an IPC. Symmetrically, the sender blocks on the receiver until it consumed all messages and replies to the sender. For example, the partners can protect themselves from being starved by using timeouts.

Since the protocol is implemented at user level, it can be adapted to incorporate specific policies regarding latency and throughput. For example, the latency of messages can be bounded by sending an IPC after the oldest message in the buffer has reached a certain age without being delivered.

This optimization does not introduce new mechanisms and thus no additional trust or protection issues. However, it is only applicable where the timing constraints of the protocol allow for the additional latency and if it is not required to send one IPC per message.

4.6.3 Sharing Meta Data

The IPC coalescing optimization increases the amount of meta data transferred with each IPC, resulting in larger cache footprint. This can be avoided by transferring meta data itself in shared memory. The IPC no longer needs to contain data and only indicates when messages are transferred by activating the receiver.

The meta data has to be interpreted by the receiver in order to access the transferred messages. When it resides in shared memory, the TOCTOU problem arises which can be addressed as discussed in section 4.6.1.

Otherwise, this optimization does not introduce additional trust or protection issues compared to IPC coalescing. It is also subjected to the same restrictions.

4.6.4 Lazy Notification

Section 4.6.2 introduced a simple protocol for managing the shared memory buffer. This protocol does not allow the receiver to consume messages which have been produced by the sender but no notification IPC has been sent for. Thus, the partner can not exploit the parallelism of shared-memory multiprocessor machines.

The protocol can be enhanced so that the receiver does not wait for an IPC from the sender. Instead, it consumes any message in the buffer based on the meta data in shared memory. Consequently, the sender and receiver can concurrently produce and consume messages. At the same time the number of IPCs and the resulting costs are further reduced.

The algorithm presented in [16] by Unrau and Krieger can be adapted to efficiently handle the cases where the shared memory buffer is empty or full. When it is empty, the receiver can indicate via the buffer that it is waiting for a notification IPC and block on the sender. When the sender produces a packet, it checks the buffer and sends a notification IPC if necessary. Symmetrically, the sender blocks on a full buffer for a notification IPC from the receiver, which is sent on the next consumed message. The protocol reduces the number of IPCs when few messages are sent or the buffer is rarely empty or full. This is the case when messages are produced and consumed at similar rates.

If the communication partners do not trust each other, they have to introduce an additional protocol, for example based on timing constraints, to avoid not being woken up by the partner.

4.6.5 Forwarding

Receivers may also act as senders to other partners. They can make use of the proposed optimizations in order to improve performance. To avoid additional trust and protection issues, they have to use separate shared memory buffers for each partner. As a consequence, forwarding messages from one of the partners to another requires to copy them from the sender's shared memory buffer to the receiver's.

A forwarder may need to modify messages that it forwards. The more a message needs to be modified, the more the costs of the modification approach those of copying it. Thus by combining both steps, the additional costs become increasingly irrelevant compared to in-place modification. This is for example the case with compression and encryption of messages.

Messages do not need to be copied for forwarding when the forwarder communicates with both partners on the same shared memory buffer. On the other hand, the protection and trust problems discussed for each optimization become transitive among the partners sharing a single buffer. Also, meta data is only valid for two communication partners and can in general

not be shared transitively. The shared resource protocols established for managing the buffer need to take the transitivity into account as well.

Chapter 5

Implementation on L4

The design presents generic solutions for achieving asynchronous IPC. Based on the experiences from constructing multi-server operating systems on L4, two scenarios are of particular interest for an implementation.

Many applications have to preserve transparency towards a synchronous protocol. At the same time, they would benefit from the properties of asynchrony. This is e.g. the case when notification messages to several other components need to be sent without blocking. Similarly, incoming synchronous request can be delivered asynchronously without requiring the receiver to block. In order to evaluate the fundamental costs for transparent asynchrony, we decided to provide an implementation which exploits all optimization possibilities presented in the design. Most importantly, the messaging system and the original component are co-located in the same address space.

For other applications, such as drivers, the achievable throughput and latency for communication are of primary importance. In particular their external protocol determines how well these requirements can be met even when crossing multiple protection domains. Thus, the second part of the implementation is guided by the intransparent optimizations discussed in the design to provide high performance to applications.

The implementation is based on the L4KA::Pistachio micro kernel which implements the L4 Version 4 API. Also, basic thread and memory management functionality of the SawMill multi-server operating system is used.

After giving a short overview on the SawMill operating system, this chapter describes our implementation focusing on transparency and throughput. The last section addresses the implementation issues of producer-consumer synchronization.

5.1 The SawMill Multi-Server Operating System

SawMill was developed at the IBM T.J. Watson Research Center focusing on the decomposition of Linux as an example for a legacy system. The resulting set of system servers is designed to maintain the semantics of the original OS kernel. In [7] Gefflaut et al. illustrate the mechanisms and protocols employed to achieve protection, coherent system semantics, and performance for the decomposed system.

An integral part of SawMill is formed by a highly flexible memory management framework described by Aron et al. in [2]. It is based on the notion of *dataspaces* as an abstract data container. It can uniformly represent arbitrary memory objects, e.g. physical or anonymous memory, files, or device related memory.

Dataspaces are implemented by *dataspace managers* which provide the dataspace contents as virtual memory. A dataspace can be accessed by associating it or parts of it with a virtual memory region in an address space (it is said to be *attached* to the region). The association is implemented by a *region mapper*, which is realized as a user-level page fault handler. It is responsible for translating an access to a particular address in an address space into a dataspace and its manager. This information is then used to request a page mapping from the dataspace manager to the address space in which the dataspace is attached.

5.2 Transparent Asynchronous IPC

In applications such as servers, asynchronous communication is often required as an internal optimization while the external protocol needs to be preserved. Our implementation addresses these requirements by providing the necessary buffer and proxy thread management functionality discussed in the design. Furthermore, it applies the presented optimizations to achieve good performance. As a library, it makes the integration of asynchronous communication into applications very simple.

5.2.1 Co-Location

The design presented several optimizations that rely on co-locating the application and the messaging system. Since our implementation is based on the IA-32 architecture, avoiding the costs of address space switches is of particular importance to provide good performance to applications.

Impersonation of threads in L4 (*propagation*) is available to threads within the same address space. Propagation across address space boundaries can be performed only by privileged threads or it is coupled with forceful redirection of IPCs via the address space in which propagation is to take place [15]. The redirection of all inter-address-space IPC through

a proxy does not only imply a significant performance impact. It can also render protocols relying on direct communication (e.g. for synchronization) ineffective. L4 supports a single redirector per address space. This prevents threads to directly communicate with multiple proxies, i.e. redirectors, in different address spaces.

Local proxy threads can handle memory mappings as part of incoming or outgoing messages in the same way as if sent directly. This simplifies the implementation compared to separate proxies that need to manage memory mappings in their address space and account clients for it.

In the L4 Version 4 API propagation is supported only for the send part of an IPC operation. Thus we implemented the scenario of sending messages asynchronously according to our design while the asynchronous receive scenario follows a worker-thread model not requiring propagation. In the following sections both cases are covered in more detail.

5.2.2 Asynchronous Send

Before sending a message asynchronously the sender needs to explicitely allocate a message buffer. It is free to do on-demand or preallocation of buffers. In our implementation the allocation is performed via the malloc() function but it can be easily adapted to other sources of buffer memory. The layout of the message buffer corresponds directly to the static L4 message registers. Thus, the contents of string items need not be copied. However, applications have to be aware that the data referenced by the string items should not be modified until the message was transferred. The buffer also contains control data for the proxy, most importantly the thread identifiers of the sender and the receiver.

The de-allocation of a message buffer is also left to the sender. Thus the proxy can communicate the status of the message back to the sender, e.g. whether the message is delivered or whether an error occurred during delivery. Instead of deallocating it, the sender can choose to re-use a message buffer.

Sending the message is implemented as adding the buffer to a message queue from which it is consumed by a proxy thread. We decided to use FIFO message ordering to preserve the message order in time. This is assumed to be a common case application requirement and serves as a base line for the performance evaluation.

Taking advantage of the optimizations presented in the design, each receiver is assigned one proxy thread. Thus, the send primitive uses per-receiver message queues. Each queue is handled by a single proxy thread. New message queues are associated with an idle proxy if one exists. Otherwise, a new proxy thread is created. Thus, the costs associated with creating a new thread occur as startup costs until the number of proxies approaches the number of busy receivers.

As shown by the design, the transparency towards existing protocols depends on the number proxy threads. On L4, the configuration of an address space determines how many proxies can reside in this address space. Thus, when transparency is required for an application, it has to ensure that its address space is configured to support a sufficient number of proxy threads.

5.2.3 Asynchronous Receive

Since L4 IPC does not support impersonation when receiving, our implementation deviates from the generic design. Most importantly, the identity of the proxy is not opaque to an actual receiver. Since the actual receiver is able to communicate with other threads impersonating the proxy, it can still achieve the impression of a single communication endpoint. Furthermore, this approach integrates well with the worker-thread model common in SawMill multi-server design.

Asynchronous reception of messages is set up by explicitely creating the proxy and an associated message queue. Each such pair is identified by a unique handle allowing threads to operate on multiple pairs of proxies and message queues. Also, any number of threads within an address space can receive from the same message queue. A receive operation dequeues a message or, if the queue is empty, optionally blocks the receiving thread for a timeout it specified. The proxy appends new messages it receives immediately to the message queue. In case there are blocked receivers, it sends a notification IPC to the thread that most recently blocked (as it has the highest probability of a still intact cache working set). Again we use a FIFO policy on the message queue and LIFO policy on the wait queue as we expect them to be most commonly required. However, any other policy can be easily implemented.

For the receive case, explicit allocation of message buffers would require the receiver to allocate buffers and pass them to the proxy for every message the proxy receives. This gives the receiver full control over the format of every single message accepted by the proxy. However, the proxy can not receive messages when the receiver does not supply buffers (e.g. because the receiver can not keep up with the rate of incoming messages). Thus, we decided on implicit on-demand allocation of message buffers by the proxy via a call back function. This mechanism provides good control over buffer allocation but de-couples the allocation from the state of the actual receiver.

The buffer returned by the call back function serves two purposes: it describes the sender(s) and the format of the messages to be accepted by the proxy and stores the contents of a received message. Before invoking the receive operation, the proxy retrieves a buffer from the call back function and loads the from specifier, acceptor, receive window, and buffer register contents from the buffer. After it received a message, it stores it in the

buffer.

Receiving string items or memory mappings asynchronously requires to set the string buffers and receive window for each receive operation so that data not yet consumed is not overwritten. Applications can implement arbitrary allocation and control policies for these secondary buffers in the call back function for buffer allocation.

5.2.4 Reply Handling

Due to the missing impersonation feature in the receive operation of L4, we can not implement reply handling as suggested in the design. An application can however achieve a similar effect for communicating with a server in RPC style by combining the send and the receive case.

First the sender has to set up a receive proxy so that it performs a closed wait on the server. Then it sends the request asynchronously to the server passing the thread identifier of the receive proxy instead of its own to the send proxy. Thus, the server replies to the receive proxy and the actual sender can retrieve the reply asynchronously from the receive proxy. Additional receive proxies are required for multiple concurrent asynchronous RPC requests.

If the server internally associates state with the thread identifiers of clients, the application has to propagate all its communication via the receive proxy to achieve full transparency.

5.2.5 Security

Our implementation does not expose memory to other protection domains thus only IPC remains a possible source of attack.

For sending messages asynchronously, proxies wait for a notification IPC if there are no pending messages to be delivered. They only accept messages from threads within their address space and are thus not vulnerable from outside.

In the receive case, worker threads wait for a notification IPC when the message queue is empty. This is a closed receive on the proxy associated with the message queue, so no other thread can send an IPC to the worker thread.

The worker threads can control the message format accepted by the proxy. They also control which senders the proxy receives messages from. Thus, the application can implement custom policies to control denial-of-service attacks.

5.3 Shared-Memory Communication

When a large amount of data, e.g. network traffic, needs to be transferred across possibly multiple protection domains, the costs of copying are prohibitive for achieving high throughput. The protocol of the data transfer is a crucial aspect for the performance of the transfer facility. Transparency to existing protocols is not an issue in such sceanrios. Our implementation relies on shared memory for message transfer and focuses on throughput and latency based on the optimizations presented in the design chapter.

5.3.1 Integration with Dataspaces

Shared memory is established in a dataspace environment by attaching a dataspace in multiple address spaces. Thus, the dataspace manager maps the memory associated with the dataspace to those address spaces. The semantics of memory contents depend on the dataspace and its manager. This setup is similar to the generic model for user level paging presented in section 4.6.1.

As a brute-force approach, a message is represented as a dataspace and transferred between two communication partners by passing access rights for the dataspace from the sender to the receiver. This generic approach has the advantage that it allows to transfer the contents of arbitrary dataspaces and thus arbitrary data between two communication partners with zero-copy semantics.

However, it requires to open a dataspace, transfer access rights, and close the dataspace for every message. Such an approach is inefficient for small messages because all steps involve requests via IPC to the dataspace manager. This incurs execution time overhead and consequential costs due to mode and address space switches.

To avoid those costs, a dataspace can be reused for multiple messages by statically sharing it between the communication partners. This optimization can be applied only to modifiable dataspaces, e.g. dataspaces representing anonymous memory. If data from another dataspace is to be transferred via the shared buffer, it needs to be attached at a different location in the sender's address space and the data needs to be copied into the shared buffer. Consequently, the zero-copy semantics are lost and the copying results in additional costs.

The copying and its costs arise because the memory mappings between the sender and the receiver are established statically. This can be avoided if the sender remaps memory dynamically to the receiver. In the dataspace framework, memory mappings for dataspaces are established by their dataspace managers. Thus, to maintain transparency to the dataspace protocol, the sender has to implement the dataspace manager interface. When the receiver attaches a dataspace exported by the sender, the sender can transfer

a message by mapping it to the receiver.

Our implementation does not make assumptions on how memory mappings are established. Thus, applications are free to implement custom policies addressing their specific requirements with regard to trust, protection, and performance.

5.3.2 Applied Optimizations

Since transparency is not relevant in this implementation scenario, the costs associated with proxy threads were avoided by eliminating the proxy threads themselves. Thus, the sender and the receiver communicate directly.

The shared memory in which the messages reside is called the *data buffer*. Meta data is also transferred via shared memory in a separate *control buffer* in order to support message forwarding across multiple address spaces in a singel data buffer. The implementation allocates meta data statically, i.e. the control buffer contains a fixed amount of message descriptors. This limits the number of messages that can reside in the data buffer. It requires applications to determine the size of the data and the control buffer according to their communication behavior before allocating them.

Messages are assumed to consist of contiguous unstructured data. Thus, a message descriptor consists of a pointer to a message and its size. Applications can layer an abstract data type on top of this representation for additional semantics. The pointer is an offset into the shared memory region, thus the shared memory does not need to be located at the same addresses in the sender's and the receiver's address spaces.

To reduce the number of necessary IPCs for message transfers, the lazy notification protocol was extended so that multiple threads on the sender and the receiver side can communicate via the data and control buffers. This made it necessary to add thread descriptors to the control buffer for managing threads that wait for notification IPCs.

The implementation allows applications to control when notification IPCs are sent to the partner and when to block on the partner to receive messages. Thus, our implementation exports policies regarding message batching and consequently transfer latency and throughput to the user.

The primitives for allocating, freeing, sending, and receiving messages operate on meta data only. The management of the data buffer needs to be performed by applications directly for maximum flexibility.

In applications where the communication partners do not trust each other the TOCTOU problem arises. It is addressed in our implementation by creating private copies of the meta data before checking and accessing it. If the memory provider, i.e. dataspace manager, of the buffers is not trusted, a protocol is necessary to recover from unhandled page faults on the buffers. On L4, such a protocol can be implemented by the pager, e.g. by requiring that a dataspace manager serves a page fault within a certain

amount of time. If this timeout expires, the faulting thread is reactivated and made to execute a handler function. This protocol can be implemented orthogonally to message transfer, hence it is not part of our implementation.

5.4 Producer-Consumer Synchronization

Section 4.6.4 discussed the lazy notification protocol for buffer management. It is based on two communicating threads which indicate via a flag in shared memory when they block and thus require a notification IPC from the partner.

If the partners do not trust each other, they can be starved on the notification IPC by a partner which sets the flag but never blocks. Thus the notification IPC has to be sent with a small or no timeout so the sender is not blocked by the receiver. If the notification IPC to the receiver fails, it is assumed to be misbehaving and the notification is dropped.

However, there are two conditions under which a notification IPC from a thread A to a thread B can fail, although B is cooperating:

- after setting the flag and before blocking, B is descheduled in favor of A, which examines the flag and sends a notification IPC that fails
- B unblocks by itself via a timeout; before it can reset the flag, it is descheduled in favor of A which unsuccessfully tries to send a notification IPC

The first case is critical because B would block without being reactivated by A. Thus it needs to be able to recover from this race condition. In the second case, it is safe to ignore the failed notification.

The delayed preemption feature of L4 allows threads to safely defer asynchronous preemptions for a certain amount of time or to generate an IPC to an exception handler thread in case an asynchronous preemption occurs. Thus, B can recover in cooperation with the exception handler when the race condition occurs. B needs to activate the delayed preemption feature before it sets the notification flag and blocks. In case B is descheduled in favor of A before it blocks, the notification IPC fails, but B can be recovered by the exception handler.

The delayed preemption feature is controlled by a privileged thread. It also depends on the priorities of threads. Since no assumptions about these entities and parameters can be made without limiting genericness, our implementation does not make use of delayed preemptions for synchronization.

Chapter 6

Results

This chapter analyzes the performance of asynchronous communication on top of the synchronous primitives provided by L4. It determines the baseline costs of the asynchronous primitives, the latency of a message transfer, the achieved throughput, and quantifies the impact of communication on application performance.

The implementation and measurements are based on a prerelease version of the L4KA::Pistachio micro kernel. The measurements were conducted on a 450 MHz Intel Pentium III processor with 16 KB 1st level cache for instructions and data each, 512 KB shared 2nd level cache, a 4 way set associative instruction TLB with 32 entries for 4 KB pages and one with 2 entries for 4 MB pages, a 4 way set associative data TLB with 64 entries for 4 KB pages and 8 entries for 4 MB pages. The machine was equipped with 196 MB RAM. We used the processor's internal performance counter registers for our measurements.

6.1 Transparent Asynchronous IPC

This chapter presents the performance of the asynchronous send primitive and compares the latency induced by an asynchronous message transfer compared to synchronous IPC.

6.1.1 Send Primitive

Figure 6.1 shows the base costs of the asynchronous send operation with and without the costs incurred by creating a proxy thread. The send primitive accepts a user-specified pointer to a message descriptor. It derives the message queue associated with the the receiver thread identified in the message descriptor via a hash table. If necessary, a new message queue is allocated. The message descriptor is appended to the message queue. In case the message queue had to be allocated, also a new proxy thread is created to handle

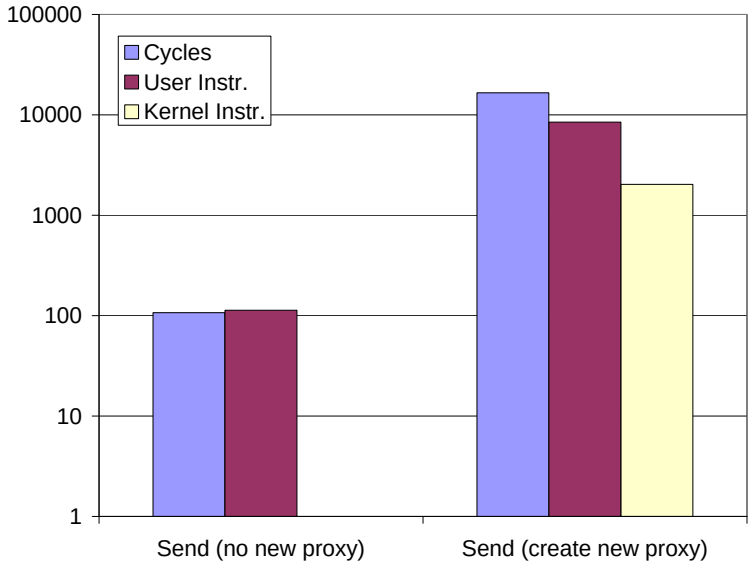

Figure 6.1: Costs of asynchronous send primitive

the queue. Otherwise, there is no interaction with the proxy thread.

The numbers reflect the average of 100 iterations as the number of cycles per message spent on the send primitive. They match our expectations of the costs for the involved data manipulations and the synchronization via a spin locks using the processor's atomic compare and exchange instruction.

Figure 6.1 also shows the overhead incurred when a new message queue and a new proxy have to be created. These costs are caused by an IPC to a privileged thread, a ThreadControl and a Schedule system call, the reply IPC, and a ExchangeRegister system call to start the proxy. The results show that thread creation in its entirety is an expensive operation, thus the optimization of pooling proxy threads is very important to mitigate these costs.

6.1.2 Latency

This measurement compares the latency induced by a synchronous message transfer with that of an asynchronous transfer via a proxy thread. In both cases, the message consists of a single word containing a cycle timestamp taken by the sender immediately before the send operation. Another timestamp is taken when the message is received. The latency is given in Table 6.1 as the result of subtracting the sender's from the receiver's timestamp. Reading and storing the processor's timestamp register takes 30 cycles which

	Messages	Latency (Cycles)	Std. Deviation
Synchronous	100000	606	8%
Async. Send	1000000	1542	6%
Async. Receive	1000000	1488	7%

Table 6.1: Per-message latency of asynchronous message transfer

is included in the results.

The asynchronous send scenario includes the allocation of a new message queue which gets associated with the receiver. A notification IPC is sent to an existing idle proxy which gets associated with the message queue. In the asynchronous receive scenario, the receiver is blocked on the proxy waiting for the message. Thus, the proxy has to send a notification IPC to the receiver to consume the message from the message queue.

This behavior models a communication protocol that requires the receiver to block on the sender until a message is received. It is common place in client / server interaction.

Sending a message asynchronously via a proxy has higher costs than when a message is received asynchronously. This is due the message queue lookup which is not necessary for receiving as there is only a single message queue.

The relatively high standard deviation of the results is due to timer interrupts occurring during the experiment. A latency of more than 600 cycles for a synchronous short IPC on L4 is surprising. However, a base-line cost of 408 cycles was determined for the measurement setup. The discrepancy of about 200 cycles is due to the overhead of timestamping and of the generic L4 convenience programming interface used in our implementation.

6.2 Shared-Memory Communication

In this section the costs of transferring messages via shared memory are presented. After analyzing the base costs of the primitives for transferring a message, the effects of crossing multiple protection domains are shown with regard to latency and throughput. The overhead of communication on checksumming as a typical operation on messages is presented and finally the overhead of copying communication is analyzed.

6.2.1 Primitives

Figures 6.2 and 6.3 show the costs of the meta data manipulation necessary to send or receive a message via shared memory. Sending a message consists of allocating a message descriptor from a pool in the control buffer, specify-

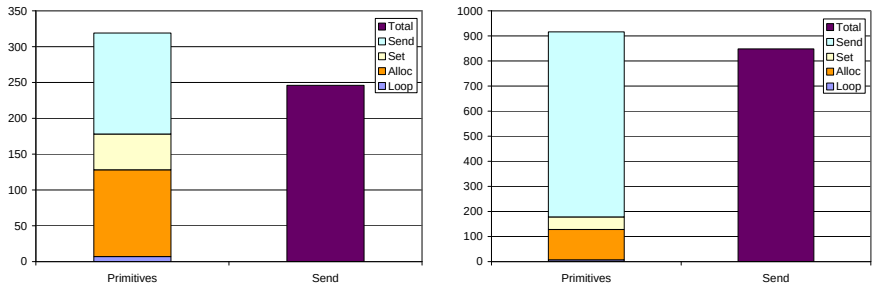

Figure 6.2: Costs of send primitives (cycles)

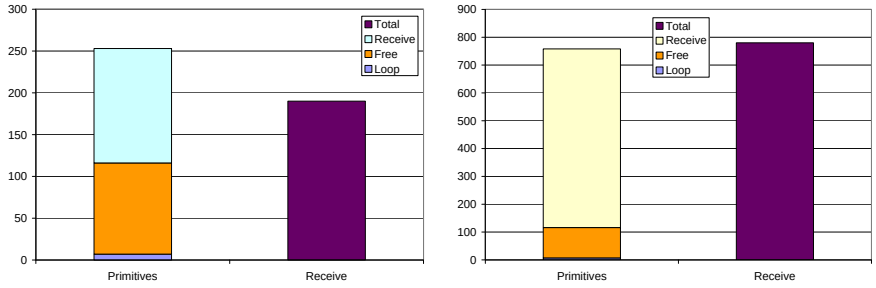

Figure 6.3: Costs of receive primitives (cycles)

Intermediate domains	0	1	2	4	8
Average latency in cycles/1000000	1.0	2.9	4.8	9.5	20.1
Throughput in packets/ms	1063	335	197	100	36

Table 6.2: Overhead of forwarding messages across multiple domains

ing the message offset and size by writing these values to the descriptor, and adding the message to the shared message queue. A message is received by removing it from the message queue, reading the message parameters, and deallocating the descriptor to the descriptor pool.

Given that pure meta data operations were measured, the costs appear to be unexpectedly high, e.g. 50 cycles for setting the message offset and size in the message descriptor. They are due to the function call overhead, pointer dereferencing, and sanity checks performed by our implementation. An optimized but less generic implementation is assumed to achieve markedly better results. Also the experiment operated on a large array of distinct message descriptors, thus it can be assumed that cache misses occurred frequently.

A notification IPC triggered by a send or receive operation increases the costs of those operations by 602 cycles for sending and 590 cycles for receiving. This is consistent with the latency measured for synchronous IPC in the previous section.

6.2.2 Crossing Multiple Domains

Table 6.2 shows the effect of crossing multiple protection domains on latency and throughput. Both values show the costs of the meta data manipulations and notification IPCs necessary to repeatedly transfer messages via a buffer with a capacity of 4096 messages. The source domain generates a notification IPC per 4096 messages and blocks for deallocated message descriptors on the receiver. Similarly, the sink domain notifies the sender of deallocated message descriptors every 4096 messages and blocks for new messages. In the intermediate domains, two different threads are used. One forwards messages towards the sink by acting both as a receiver and a sender. The other one forwards deallocated message descriptors towards the source. The latency is measured as in section 6.1.2 between the source and the sink and the average over all messages is shown.

The high average latency in the base case without an intermediate domain results from the fact that the transfer of 4096 messages is batched. Only after the sender sent all 4096 messages, the receiver is activated.

Dividing the average latency by the number of messages yields per-message transfer costs of 244 cycles. This value roughly reflects the costs of transferring a message. However, a message transfer requires both the sending and the receiving primitives to be invoked. Thus, based on the costs

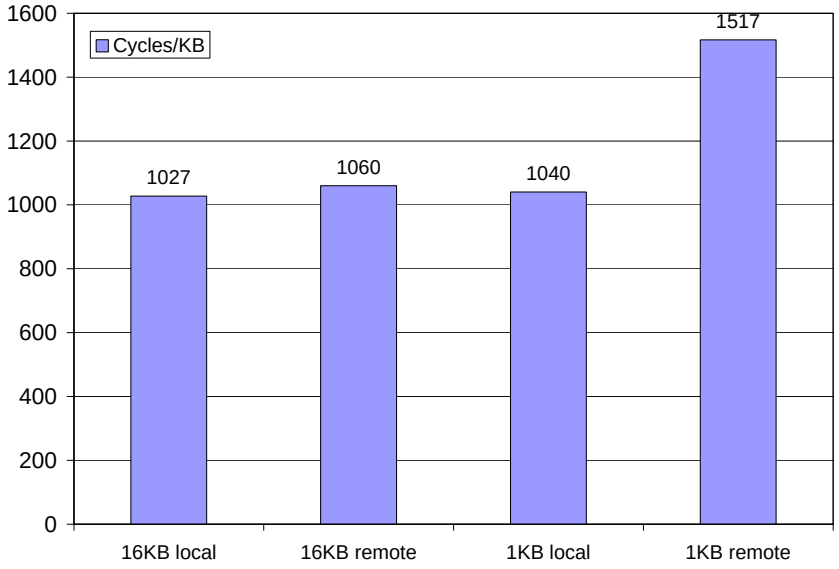

Figure 6.4: Message transfer overhead on checksumming

determined for the primitives, twice as many cycles would be expected. We assume that the hardware caches were better utilized than in the previous experiment because the 4096 message descriptors form a significantly smaller working set (80 KB).

The non-linear performance reduction per intermediate domain is not unexpected. One reason for the non-linearity are the increasing consequential costs of address space switches. Another reason is that in addition to the message transfer from the data source to the sink, deallocated message descriptors have to be passed in the opposite direction.

6.2.3 Communication Overhead

This experiment determines the overhead of the communication primitives on application performance. As a typical operation on data the TCP checksum algorithm [4] was chosen. For the measurement a data buffer is partitioned into 256 messages and the TCP checksum is calculated for each message. We compare checksumming with and without invoking the receive and de-allocation primitives that potentially blocking the receiver. The sender uses the data buffer as is and does not access it to transfer the messages repeatedly to the receiver. The data buffer is statically shared between the sender and the receiver. The checksumming is performed in the receiver after receiving a message from the sender. No page faults occur. In total,

Message size in bytes	IPC copy overhead
4	0.922
8	0.925
64	1.217
1024	1.944
4096	5.828

Table 6.3: Overhead of message copying

4 GB of data is checksummed and the results are given as the number of cycles consumed per kilobyte of checksummed data.

As depicted in Figure 6.4, the overhead incurred by receiving the messages via the data buffer is 3% for 16 KB messages. However, with 1 KB messages the overhead increases to 46% compared to checksumming without invoking the communication primitives. There are two major reasons for this behavior:

- with 1 KB messages the communication primitives need to be invoked 16 times more often per KB data to be checksummed than with 16 KB messages.

- a pipelining effect is introduced by kernel scheduling for 16 KB messages. The receiver can checksum at least one but not all messages in the buffer within its timeslice. Thus, the receiver sends a notification IPC in case the sender is blocked and continues checksumming until it is descheduled. The sender reuses the message descriptors freed by the receiver to send more messages. They are picked up by the receiver without ever blocking on an empty message queue. For 1 KB messages however, the receiver frequently blocks on an empty message queue requiring notification IPCs from the sender.

6.2.4 Copying Overhead

Table 6.3 compares the cycles required to transfer messages via IPC and via shared memory using our implementation. The shared memory control buffer contains 16 message descriptors. For the measurement, 16 messages with their sizes ranging from 4 to 4096 bytes are transferred. This includes the notification IPC of the sender to the receiver sent with the first message. For the IPC scenario, a single IPC is sent transferring 16 indirect strings each of which references data of the given sizes. The overhead is given as $(cycles\ for\ IPC)/(cycles\ for\ shared\ memory)$.

This setup is not modeled after a realistic communication model. The performance of the synchronous IPC could for example be increased by

transferring the messages as direct instead of indirect strings. The results are meant to roughly quantify the overhead caused by copying during a message transfer.

Chapter 7

Discussion and Interpretation

This section evaluates the benchmark results presented in Chapter 6. It discusses their implications on the management of proxy threads, on system design with regard to the nesting depth of communicating components, and on the importance of application specific policies for performance.

7.1 Proxy Management

Creating a proxy thread for asynchronous communication is significantly more expensive than the message transfer itself. This is shown by the benchmarks of the asynchronous send primitive. To achieve good performance for asynchronous communication, it is very important to reduce the number of necessary proxy threads. The design discusses several optimizations addressing this problem.

Our implementation creates new proxy threads on demand. If this demand arises, a message transfer is delayed until a new proxy is created. This can violate protocols with rigid timing constraints. Thus, this approach is not applicable for preserving transparency to such protocols. In order to satisfy the timing constraints, proxy threads have to be pre-allocated.

In case an existing proxy is assigned a message transfer and the destination thread is ready to receive, the message latency is increased by 154% compared to a synchronous transfer. Given the base-line costs of 185 cycles for intra address space IPC and 408 cycles for inter address space IPC, at least 45% overhead can be attributed to the notification IPC. Lazy process switching, which was not available in L4KA::Pistachio at the time of the measurements, is expected to reduce this overhead significantly for receiving messages via a proxy.

The best case behavior allows a sender to send a message asynchronously with 75% less execution time overhead than an empty synchronous IPC. This

behavior is achieved for example with bursty communication characteristics.

7.2 Modularization

A substantial performance degradation of crossing multiple protection domains with our implementation is shown in section 6.2.2. This limits the nesting depth of components transferring messages on a data path at which acceptable performance can be achieved. Four intermediate domains can be seen as a landmark figure for which latency is increased and throughput reduced by an order of magnitude.

The recursive deallocation of message descriptors via the intermediate domains was identified as one source for these costs. It is not possible to deallocate the descriptors for a region in the data buffer globally in our implementation because the message descriptors are shared privately between each pair of communication partners and are not directly related to each other.

To avoid the costs of recursive deallocation, the meta data needs to be either shared globally which implies transitive trust relationships between the involved communication partners. Alternatively, the message handling and the meta data can be centralized in a protection domain similar to the fbufs approach.

7.3 User-Level Policies

The checksumming experiment illustrated that the costs of common operations on messages can cover the overhead induced by communication. The flexibility of the primitive operations also allow applications to benefit from their low costs by implementing custom policies.

Another common operation on messages is to add or to remove protocol headers. These operations can be performed with zero-copy semantics by layering an abstract data type on top of our implementation and allocating additional message descriptors on demand.

Similarly, the pipelining effect observed in the checksumming experiment is achieved by the specialized message handling policy of the receiver. Thus, it is essential to export the freedom to apply custom policies to applications. This allows applications to benefit from additional semantics and increased performance.

Chapter 8

Conclusion

IPC is a central mechanism for constructing componentized systems as it allows components to safely interact across the protection domains they reside in. Two fundamental IPC models are synchronous and asynchronous communication. The L4 micro kernel shows that the synchronous blocking and unbuffered model can be implemented very efficiently. The asynchronous model provides non-blocking and buffered communication and thus parallelism but is inherently associated with policy, e.g. for the buffer management.

Since both models can be emulated by each other, it is desirable to combine their advantages. To benefit from high IPC performance and exporting policies to applications, we propose to emulate asynchronous communication semantics on top synchronous IPC primitives.

The approach of this thesis is to buffer messages and to achieve parallelism via additional proxy threads at user level. The design focuses on trust, protection, transparency, and performance as key aspects. It illustrates how asynchronous semantics can be transparently and safely interposed between communication partners while preserving their existing communication protocols. For components requiring asynchronous communication while maintaining transparency towards an external protocol several performance optimizations are presented.

In scenarios demanding high throughput for cross domain data transfer the costs of copying data across protection boundaries is prohibitive. Thus, the design provides asynchronous communication with zero-copy semantics by transferring messages via shared memory. With additional optimizations it addresses the costs caused by IPC system calls and address space switches to further reduce the communication overhead.

We show that asynchronous communication can be implemented on the L4 micro kernel purely at user level on top of its synchronous IPC primitives. In the common case, the asynchronous primitives for transparent communication via proxy threads achieve performance comparable to the

synchronous primitives. In the best case, they outperform the synchronous primitives by a factor of four. The message transfer via shared memory is shown to effectively eliminate the overhead of copying and to provide high throughput.

The results also highlight the importance of providing a high degree of flexibility to applications. It allows to implement application specific policies for additional semantics and increased performance. A micro kernel with flexible and well-performing synchronous communication primitives such as L4 has been found to be an ideal basis for achieving this goal.

The obtained performance encourages the construction of modularized systems. However, the performance impact of decentralized message handling limits the number of domains that can be crossed efficiently with our zero-copy approach. This impact has to be reduced for fine-grain componentized systems and constitutes an area of further research.

Chapter 9

Future Work

This chapter gives an overview on topics that could not be covered in this work due to the limited amount of time available.

9.1 Multi-Processor Support

On uniprocessor systems, the parallelism provided by asynchronous IPC increases concurrency but does not necessarily result in an overall reduction of execution time. This effect can be achieved on multi-processor machines.

However, thread migration and cross-processor communication can incur a significant overhead. Furthermore, applications and proxy threads co-located on a single processor might benefit from utilizing shared local processor caches. Pipelining effects as exposed by the checksumming experiment in section 6.2.3 are also expected to become even more important on multi-processor systems. Thus, an analysis of the communication behavior on multi processors and the involved trade-offs is necessary in order to maximize the benefit provided by the additional parallelism.

An important aspect on multi processors is synchronization. Coarse-grained pessimistic synchronization is currently used in our implementation. This is acceptable for uniprocessors where experiments have shown that up to 16 threads involved in shared memory communication have no visible effect on message latency and throughput. With 256 involved threads the overhead due to synchronization is 30%.

However, the simple synchronization is assumed to hamper performance on multi processor systems where lock contention becomes more likely. Improvements can be expected from well-known optimistic fine-grained synchronization techniques. An evaluation of the applicability of the delayed preemption mechanism available on L4Ka::Pistachio is of particular interest.

9.2 Analysis of Cache Impact

The cache behavior of communication facilities is crucial for performance. For example, Mach IPC performance suffers significantly from the cache footprint of the Mach kernel, as we discussed in section 3.1.

Thus, an emulation of asynchronous IPC can only deliver maximum performance to applications by optimizing it for low cache impact. An analysis has to show under which circumstance our implementation exposes high cache footprint and identify possible solutions. In particular for asynchronous communication via proxies, we expect that e.g. a reduction of the meta data size and buffer reuse are possible methods of improving cache behavior.

9.3 Access Revocation on Shared Buffers

The TOCTOU problem discussed in section 4.6.1 is addressed in our implementation by copying the possibly volatile data to private memory before checking and accessing it. Due to the small size of the meta data that is copied, the performance impact is acceptable. However, in applications that need to check and access large messages, the overhead introduced by this approach can be prohibitive.

Dynamic revocation of access rights on the memory to be accessed was discussed as an an alternative to copying the data. The dataspace model implemented in SawMill supports this solution. However, it requires the receiver to contact the dataspace manager via IPC and is thus also associated with inherent costs. Another possible solution is to implement the protection semantics in the dataspace manager providing the buffer memory and to enforce the access restrictions on demand when memory mappings are requested by the involved communication partners.

An analysis of these approaches has to show under which circumstances which method is most applicable with regard to trust, protection, and performance.

9.4 Impact of Intermediate Domains

The benefits of modularization and encapsulation speak in favor of systems with a very fine-grained level of componentization. However, the costs of communication across multiple protection boundaries increases significantly with each intermediate domain as shown in section 6.2.2. We attributed the costs mainly to the mechanism of deallocating message descriptors and the impact of address space switches.

Analyzing these costs in more detail can help to identify their particular sources. Based on such an analysis, specific improvements could be devel-

oped for both the process of switching address spaces and the behavior of the communication system.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceeding of the USENIX 1986 Summer Conference*, pages 93–112, Atlanta, 1986. USENIX.

[2] Mohit Aron, Jochen Liedtke, Yoonho Park, Luke Deller, Kevin Elphinstone, and Trent Jaeger. The SawMill framework for virtual memory diversity. In *Australasian Computer Systems Architecture Conference*, Gold Coast, Australia, January 2001. IEEE Computer Society Press.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, 1991.

[4] B. Braden, D. Borman, and C. Partridge. Computing the internet checksum; RFC 1071. *Internet Request for Comments*, (1071), September 1988.

[5] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[6] B. Ford and J. Lepreau. Microkernels should support passive objects. *Systems*, pages 226–229, December 1993.

[7] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[8] J. Liedtke and H. Wenske. Lazy process switching. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 15–20, 2001.

[9] Jochen Liedtke. Improving ipc by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, December 1993.

[10] Jochen Liedtke. On micro-kernel construction. In *Symposium on Operating Systems Principles*, pages 237–250, 1995.

[11] M. J. Karels M. K. McKusick, K. Bostic and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Longman, Inc., 1996.

[12] J. C. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.

[13] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[14] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 170–185. Kiawah Island Resort, near Charleston, Sout Carolina, December 1999.

[15] L4Ka Team. *L4 Experimental Kernel Reference Manual*. System Architecture Group, University of Karlsruhe, May 2003.

[16] R. Unrau and O. Krieger. Efficient sleep/wake-up protocols for user-level IPC. Technical report, Dept. of Computer Science, University of Alberta, Edmonton, Canada, 1997.