TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Inter-Process Communication

Björn Döbel

Dresden, 2008-11-04

So far...

- Microkernels
- Basic resources in an operating system
  - Tasks and Threads
    - Execution contexts
    - Spatial isolation through virtual memory
    - Scheduling
    - Synchronization
  - Memory
    - Hierarchical memory management in user space
    - L4: dataspaces, region management

Today

- Inter-Process Communication (IPC)
  - Purpose
  - Implementation
  - Tool/Language support
  - Security – Who speaks to whom?
  - Safety – Communicating reliably
  - Optimizations

Why do we need to Communicate?

- IPC is a fundamental mechanism in a µ-kernel-based system:
  - Exchange data
  - Synchronization
  - Sleep, timeout
  - Wait for HW interrupt
  - Grant access to resources (memory, I/O ports, ...)
  - Exceptions
- Liedtke: *"IPC performance is the master."*

Exploring the Design Space

- Asynchronous IPC (e.g., Mach)
  - "Fire and forget"
  - In-kernel message buffering
  - Two problems:
    - Data copied twice
    - DoS attack on kernel memory (never receive data)
- Synchronous IPC (e.g., L4)
  - IPC partner blocks until other one gets ready
  - Direct copy between sender and receiver
  - E.g., Remote Procedure Call (RPC)

L4 IPC - Basics

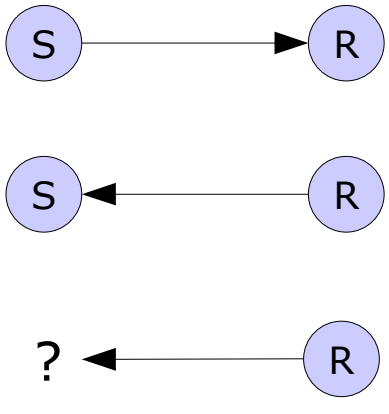
- Basic data types:
  - Bulk data (short / direct string IPC)
  - Memory references (indirect string IPC)
  - Resource mappings (flexpages)
- Types
  - Short, direct long, indirect long
  - Send, receive, receive_any, call, reply_and_wait

L4 IPC – Advanced Features

- Timeouts
  - 0 (non-blocking IPC)
  - NEVER or specific value – block until partner gets ready or timeout occurs
  - sleep() is implemented as IPC to NIL (non-existing) thread with timeout
- Exceptions
  - Certain conditions need external interaction
    - Page faults
    - L4Linux system calls
    - Virtualization faults (-> lectures on virtualization)

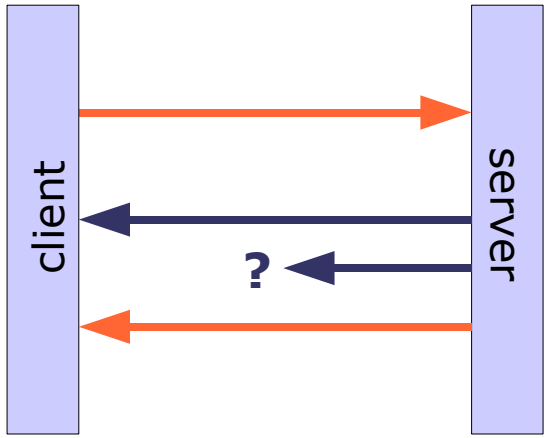
L4 IPC Flavors

Basics



- Why is there no broadcast?

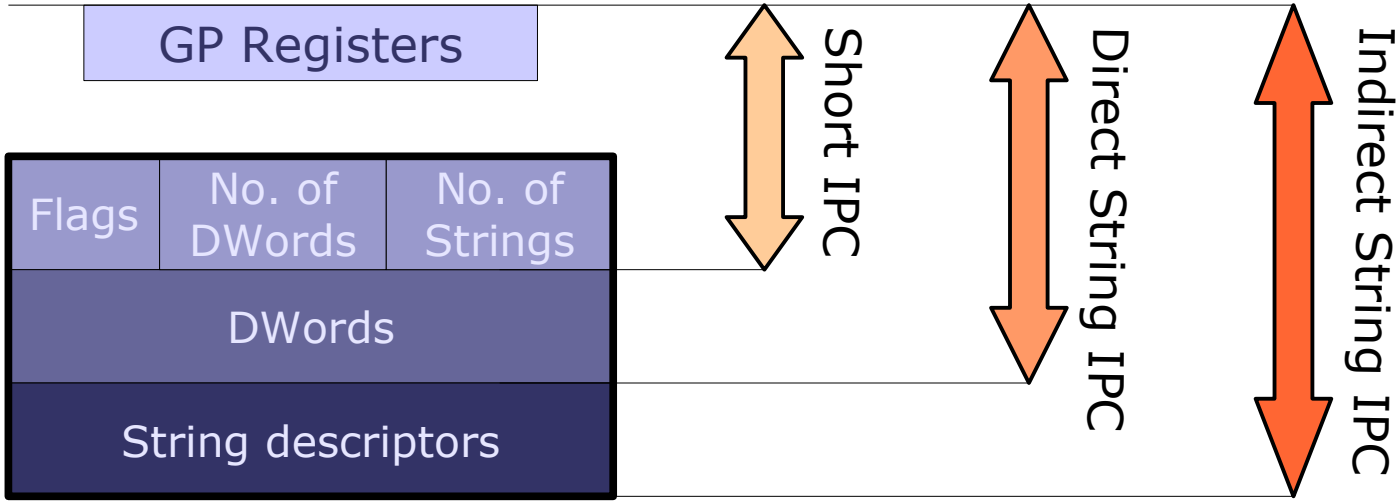
Special cases for client/server IPC



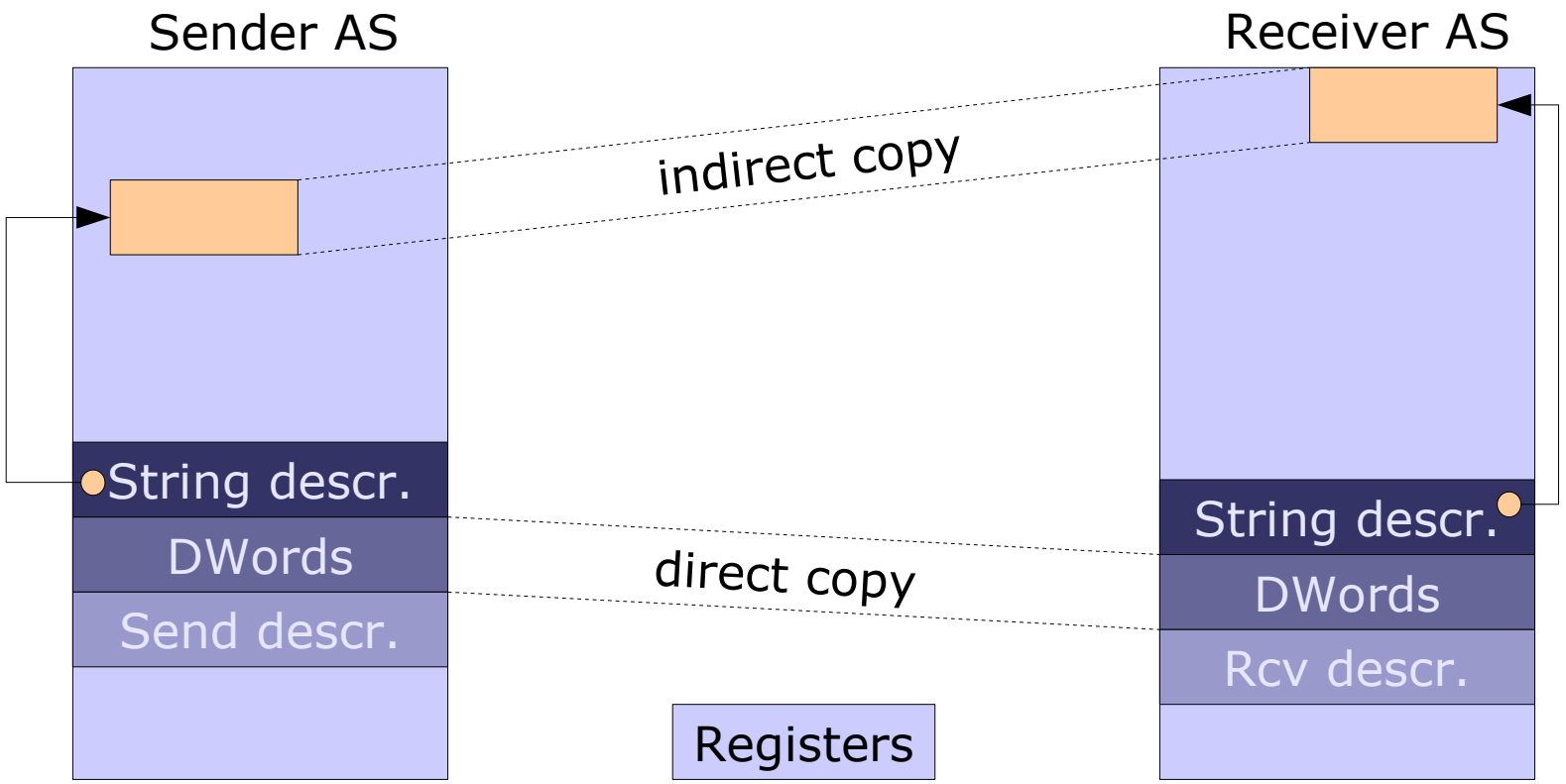
- **call** := send + recv from
- **reply and wait** := send + recv any

L4 IPC – Message Descriptor

- Send/recv operations are described by a message descriptor
  - Flags: message state (e.g., fpage included?)
  - Dwords: directly copied data
  - String descr.: pointers to indirect buffers



L4 IPC Types



Short IPC: no copying, just switch to receiver and leave register contents as is

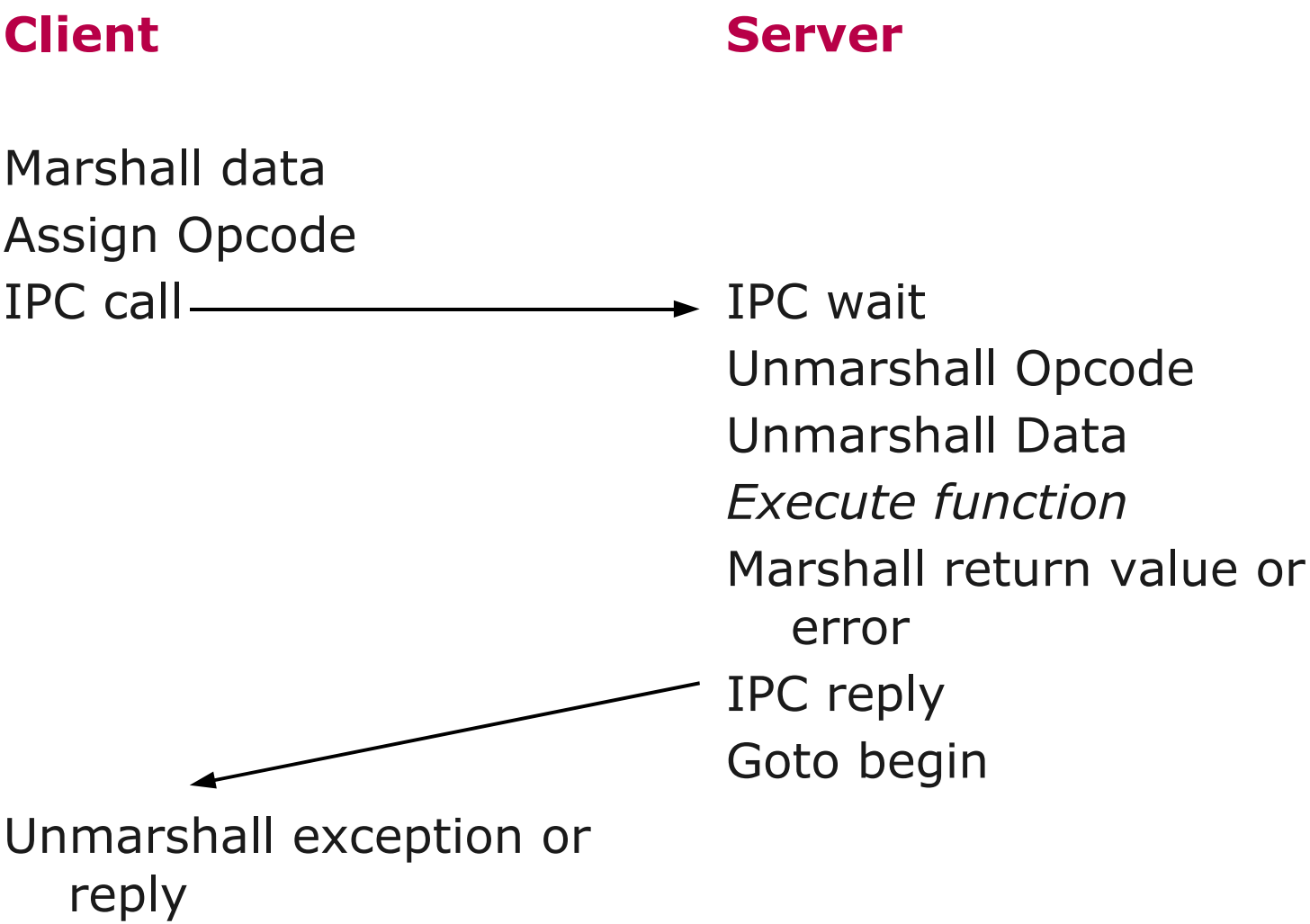
(In)direct String IPC: copy data according to msg descriptor

Break

- IPC purposes
- IPC flavors
- IPC types

- Now: Using these primitives in applications

Client-Server RPC Broken down



Writing IPC code Manually

- Setup message descriptor
  - Opcode and parameters end up in a single message buffer
  - Client and server take care of (un-)marshalling
  - Tricky: complex data types
    - Need to be flattened (pointers are useless across address spaces)
  - Highly performant code
  - Code is hard to read, write, and maintain.
- Issue L4 system call

How About Automation?

- Specify the interface of server in *Interface Definition Language* (IDL)
  - High-level-language
  - Easy to read/write
- Use IDL Compiler to generate IPC code
  - Automatig assignment of RPC opcodes
  - Generated marshalling/unmarshalling code
  - Built-in error handling
  - Server function templates to fill in
- For L4: Dice – DROPS IDL Compiler

IDL vs. Manual code

- Hard to write IPC code manually
- Use of high-level language and IDL compiler makes things easier
- Additionally:
  - Type checking: generated code stubs make sure that client sends the correct amount of data, having proper types
  - IDL compiler can optimize code
  - Use IDL interfaces to generate
    - Documentation
    - Unit tests
    - ...

Exploiting C++ Language Features

- Client:

```
ipc_stream << Opcode::ADD << 2 << 2 <<
  Genode::IPC_CALL >> result;
```

- Server:

```
ipc_istream >> opcode;
switch(opcode) {
   case Opcode::ADD:
      ipc_istream >> a >> b;
      ipc_ostream << (a+b);
      break;
   default:
      panic("unknown opcode");
}
```

Dynamic RPC Marshalling in Genode

- C++-based operating system framework
- Abstract from the underlying kernel
  - Runs on Linux and L4.Fiasco
  - IPC mechanisms differ (built-in mechanism in L4.Fiasco vs. UDP sockets in Linux)
- Communication abstraction: IPC streams
  - Use C++ templates to allow writing arbitrary (*primitively serializable!*) objects to IPC message buffer
  - Special values (Genode::IPC_CALL) lead to calls to underlying system's mechanism

DynRPC Summary

- Major advantage:
  - Very small implementation making use of suitable host-language features
  - C++ compiler can heavily optimize IPC path
- No automatic (un)marshalling
  - Use whatever serialization mechanism you like
- No builtin type checking
  - Developer needs to care about amount, type and order of arguments
- Orthogonal to use of IDL compiler
  - Generate IPC stream code from C++ class definitions (Liasis IDL compiler, work in progress)

Break

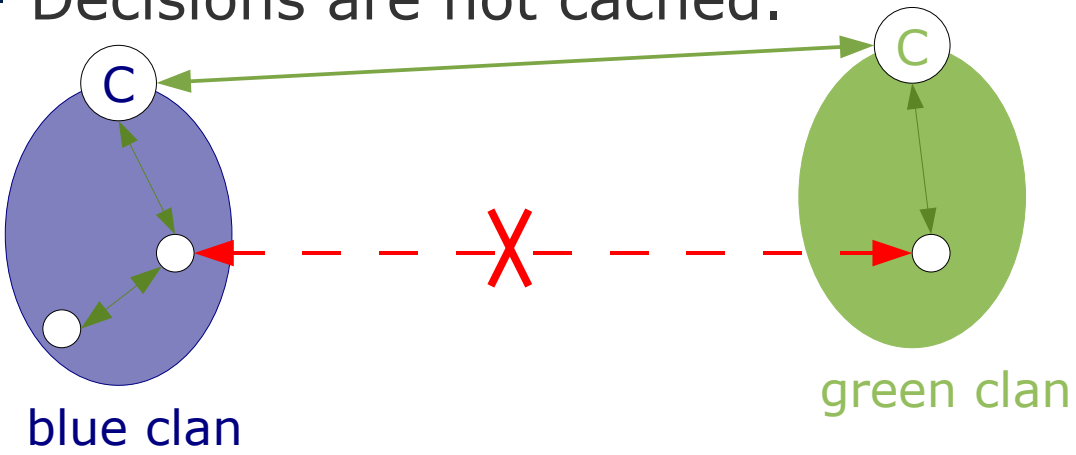
- Client-server RPC
  - Opcodes
  - Marshalling
- Use of programming language features
- Next step: Secure communication

IPC & Security

- Problem: How to control data flow?
- Crucial problem to solve when building real systems
- Many proposed solutions

L4v2: Clans & Chiefs

- Tasks are owned by a chief.
- Clan := set of tasks with the same chief
- No IPC restrictions inside a clan
- Inter-clan IPC redirected through chiefs
- Performance issue
  - One IPC transformed into three IPCs
  - Decisions are not cached.

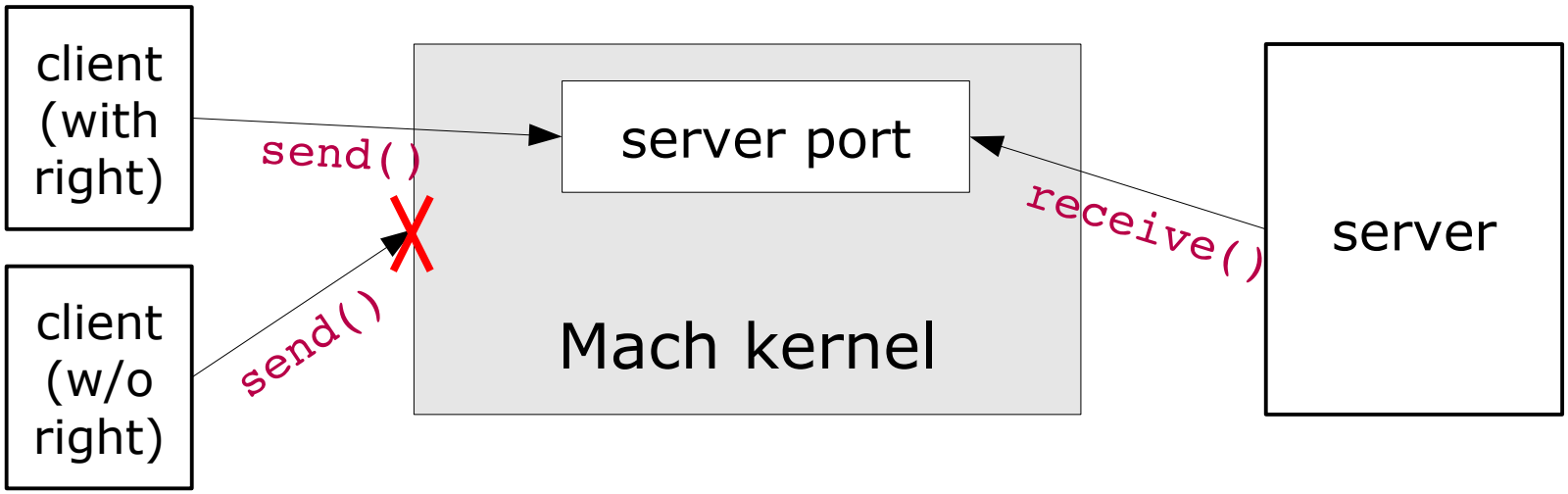


L4/Fiasco: Reference Monitors

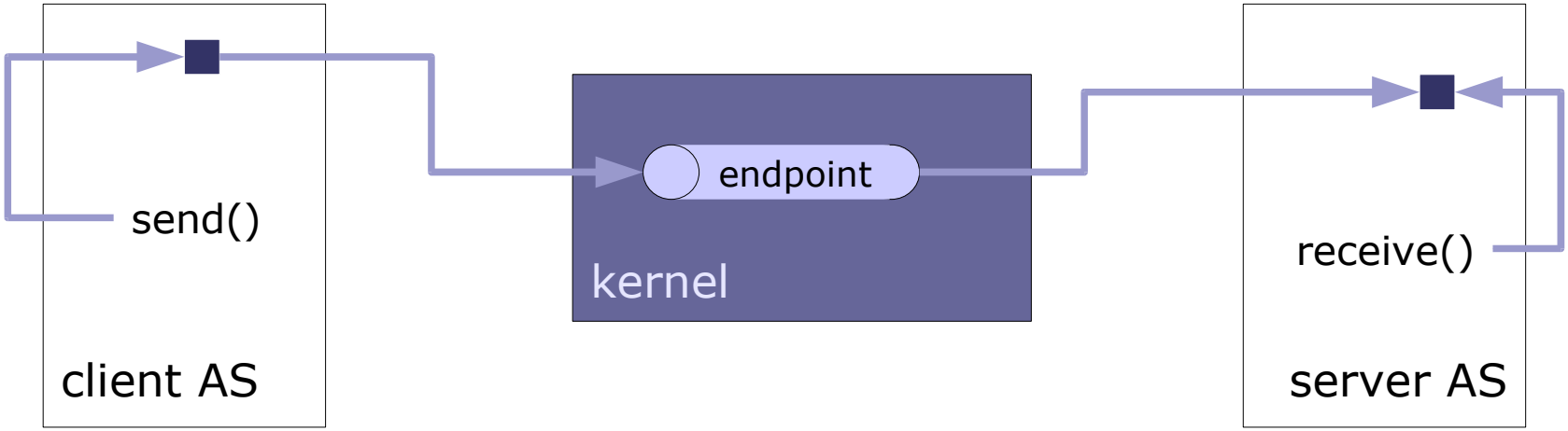
- New abstraction: communication is allowed if certain flexpage has been mapped to sender
- Every task gets a reference monitor assigned.
- Communication:
  - IPC right mapped?
    - Yes: perform IPC
    - No: raise exception at reference monitor
  - Reference monitor can answer exception IPC with a mapping and thereby allow IPC
- Fine-grained control
- No per-IPC overhead, only one exception in the beginning

Mach: Ports

- Dedicated kernel objects
- Applications hold send/recv rights for ports
- Kernel checks whether task owns sufficient rights before doing IPC



- Idea:
  - Invoke IPC on a kernel-object (like Mach ports, *but memory for allocation is provided by task*)
    -> endpoint (capability)
  - Kernel object mapped to a virtual address – task only knows object's local name -> no information leaks through global names

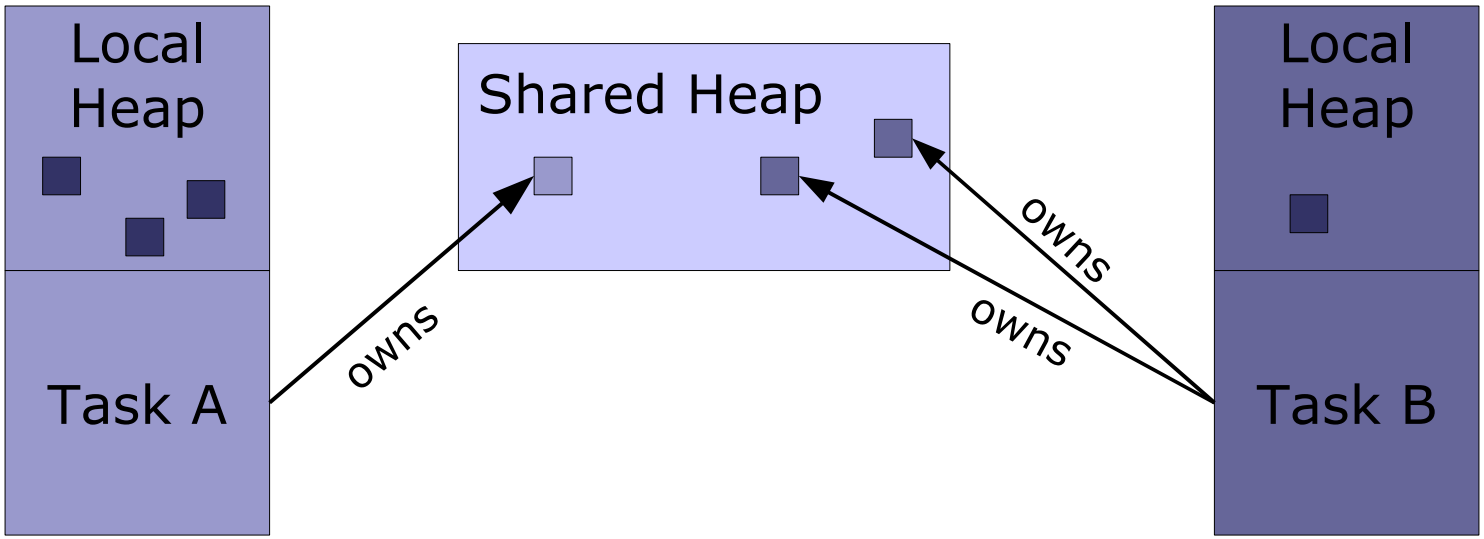


Singularity

- Singularity
  - Research microkernel by MS Research
  - Written in a dialect of C# (Sing#)
  - Topic of a paper reading exercise
- All applications run in privileged mode.
  - No system call overhead – syscalls are real function calls
- Enforce system safety at compile time.
  - Isolation completely realized using means of the used programming language -> Language-Based Isolation

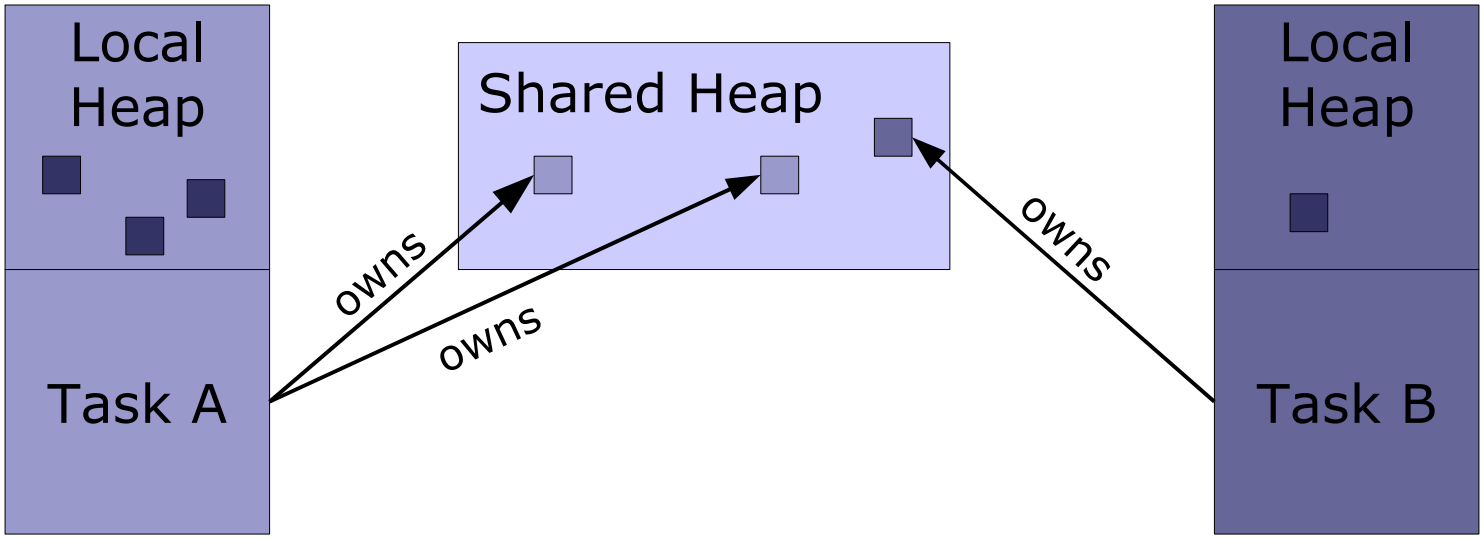
IPC & Language-Based Isolation

- Singularity IPC is always performed through shared memory.
- Only certain objects can be transferred.
  - Allocated from a special memory pool
    -> shared heap



IPC & Language-Based Isolation (2)

- Only one task may own objects in SH.
- IPC := transfer ownership of an object in SH.
- IPC protocols are specified by state machines – contracts
- Contracts are verified at compile-time



Break

- Mechanisms for controling information flow
  - Special IPC control mechanism (traditional L4)
  - Reuse other kernel mechanism (e.g., mapping of memory pages) for IPC control (L4.Fiasco)
  - Special kernel objects for IPC (Mach, L4.Florence, L4Re)
  - Static compile-time analysis of communication behavior (Singularity)

Asynchronous IPC & Shared Memory

- Some applications need high throuput for a lot of data.
  - Sharing memory between tasks can provide better performance
- Many workloads need asynchronous communication.
  - Can be built on top of synchronous IPC

Shared Memory

- Zero-copy communication
  - Producer writes data in place
  - Consumer reads data from the same physical location
- Kernel seldom involved
  - At setup time: establish memory mapping (flexpage IPC + resolution of pagefaults)
  - Synchronization only when necessary
- Ergo: Shared mem communication is fast (if the scenario allows it)
  - High throughput, large amount of data
  - Example: streaming media applications

Example: Consumer-Producer Problem

- Shared buffer between consumer and producer
- Wake up notifications using IPC
  - If new data for consumer is ready
  - If free space for producer is available

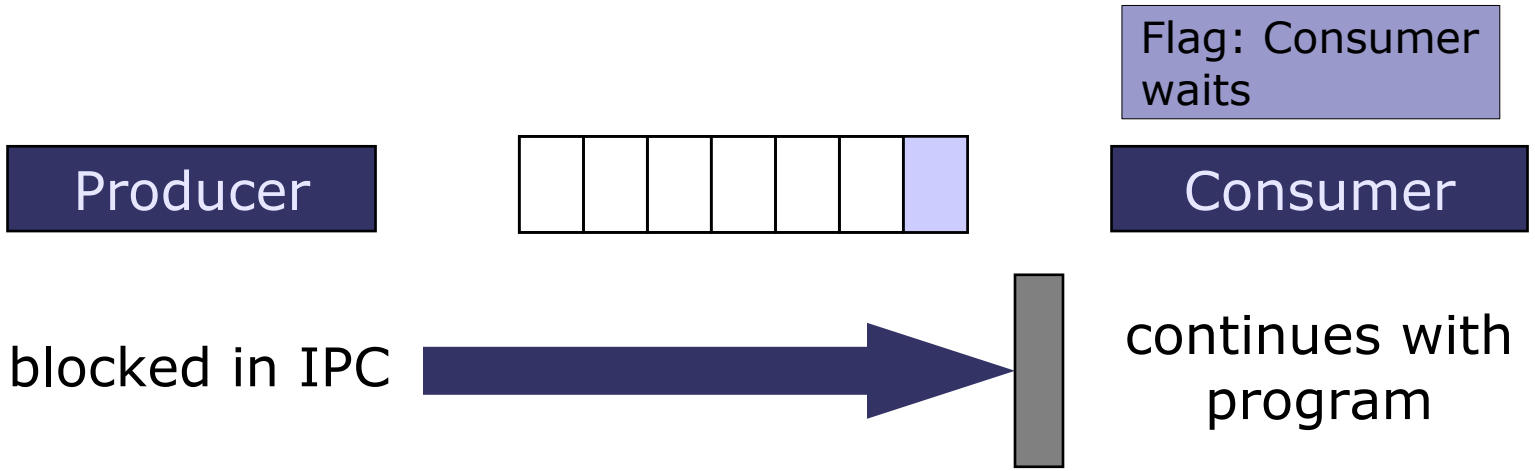
generate data (recv from network, keyboard events, ...)

FIFO queue

process data

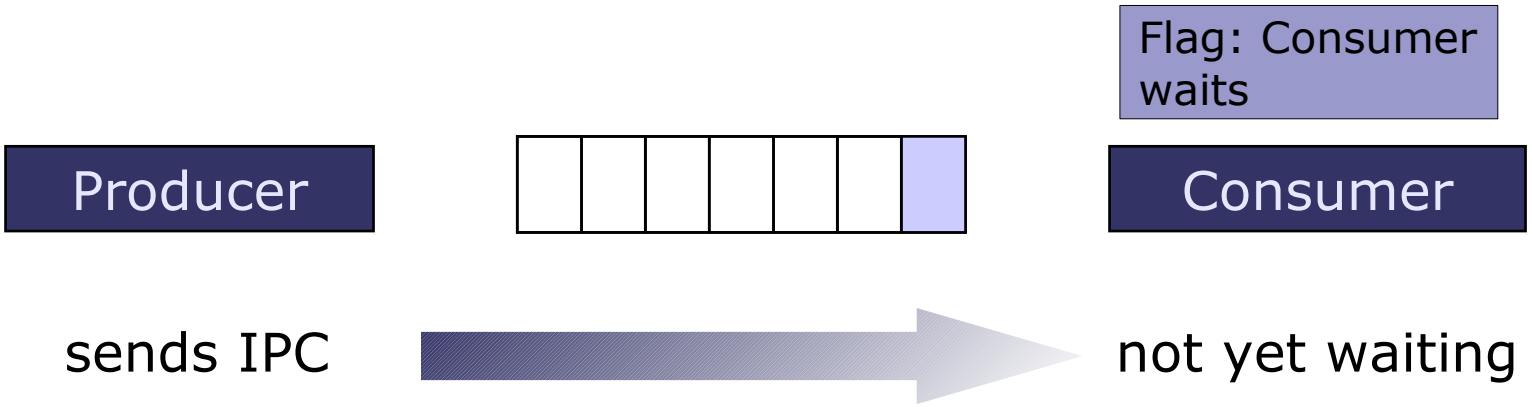
1st try: Consumer sets flag

- Consumer indicates “I am ready to receive.” using a flag and waits for IPC
- Producer sends notification IPC with infinite timeout
- Evil consumer: sets flag, but doesn't wait
- Producer remains blocked forever -> DoS



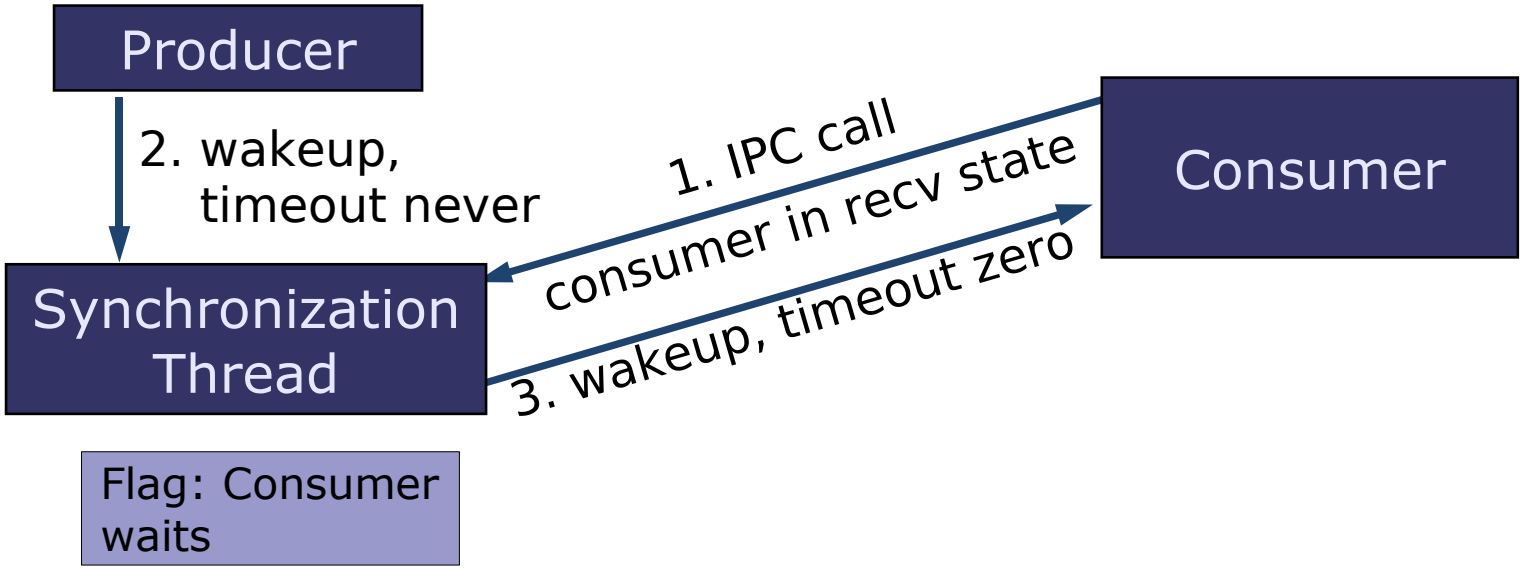
2nd try: Notify with zero Timeout

- Consumer flags “I am ready.”
- Producer sends notification with timeout zero
- Consumer in bad luck: sets flag and gets interrupted right before waiting for IPC
- Producer sends notification
- Consumer is blocked forever



The Problem: Atomicity

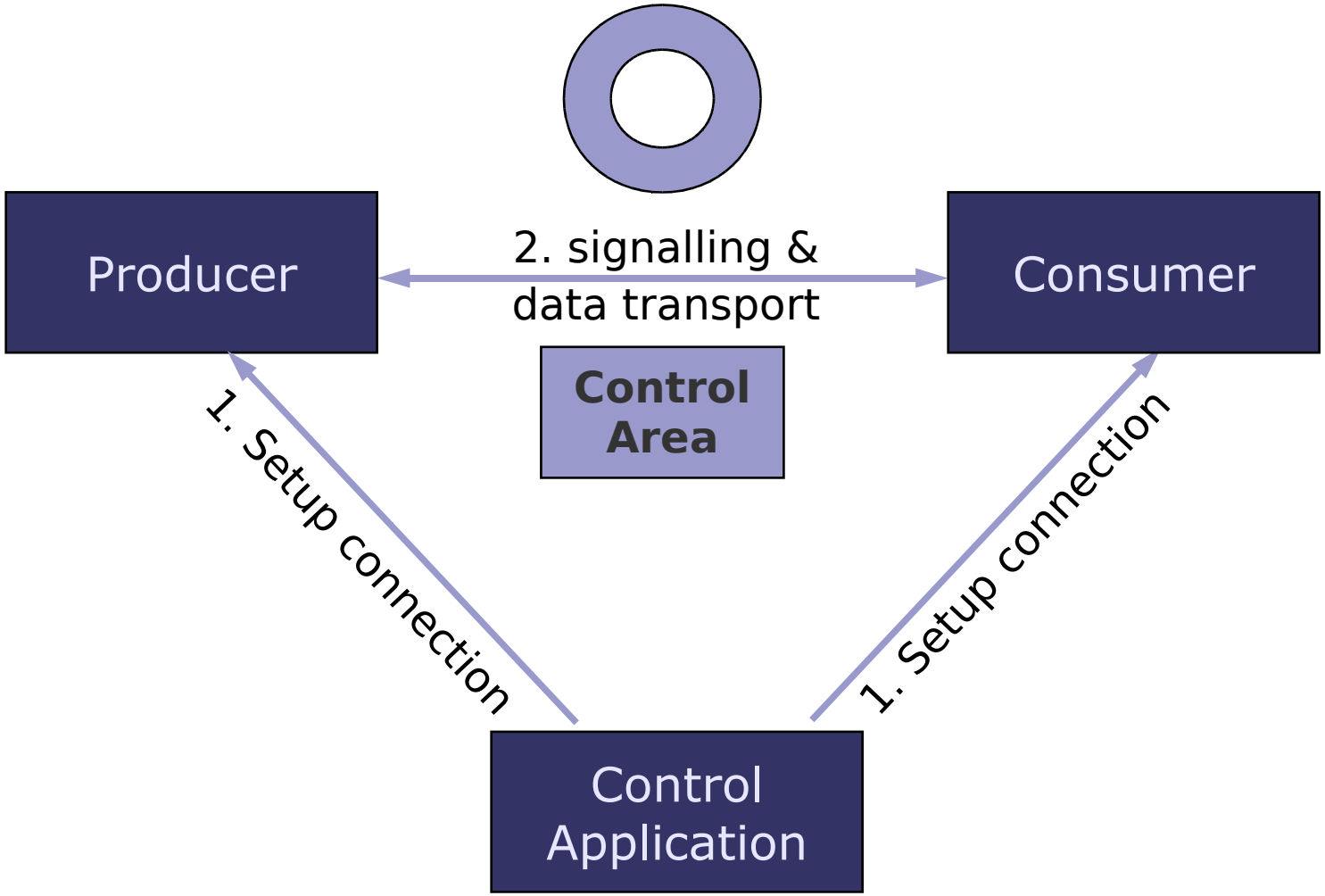
- Real solution: set flag and enter wait state atomically
- (Delayed preemption)
- L4 IPC call is atomic



DROPS Streaming Interface (DSI)

- User-level timed packet-oriented zero-copy transport protocol
- Consumer/producer scheme on a shared ring-buffer
  - Buffer holds packet descriptors for flexible indirection
- Real-time capable
  - Jitter-constrained periodic streams
  - Maths involved allow to specify an upper bound on the jitter for consumer and producer by calculating the correct dimensions for shared buffers.
  - No data loss

DSI Application Layout



DSI: Sending data

- Protocol:
  - Get a free packet from ring buffer
  - Fill in data
  - Commit (and notify producer if necessary)
- Packets
  - Describe a memory area (address + size)
  - May be
    - in a shared dataspace
    - in I/O memory
    - physical addresses

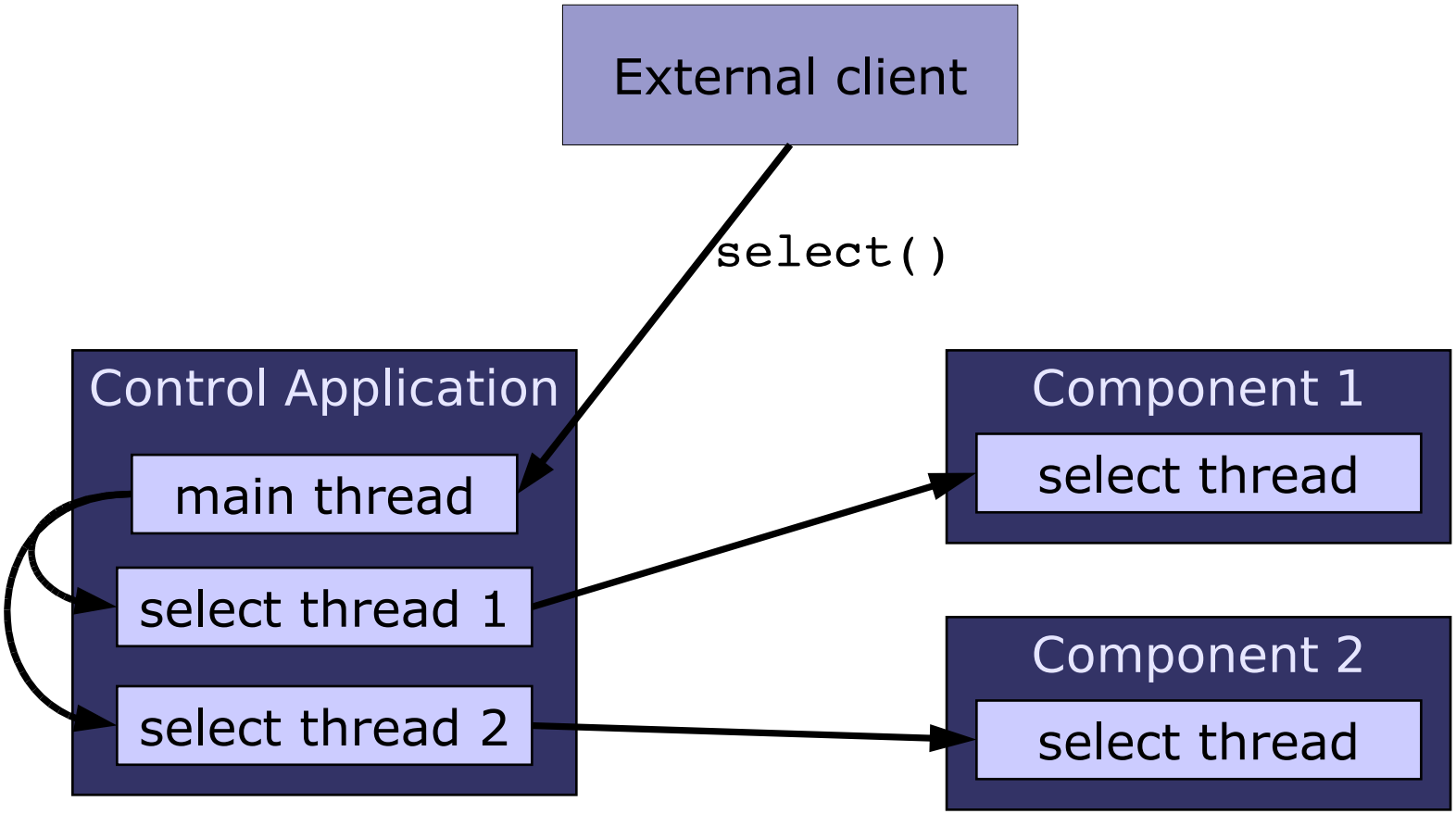
DSI: Receiving data

- Protocol:
  - Producer retrieves packet from ring buffer
  - Access data
  - Re-commit to buffer for next use by the client
- Consumer must not access packet after re-commiting!

DSI: select()

- Control app provides means to perform select
- Waits for events from multiple sources
  - May come from producers or consumers
- Cannot use single-threaded synchronization
  - We need to wait for multiple threads at once.
- Solution:
  - Use one select thread per available event source
  - Main thread waits for notification from select threads

DSI: select()


External client
select()
Control Application
main thread
select thread 1
select thread 2
Component 1
select thread
Component 2
select thread

Further Reading

- L4 kernel manual:
  http://l4hq.org/docs/manuals/Ln-86-21.pdf
- Dice manual: http://os.inf.tu-dresden.de/dice/manual.pdf
- Genode Dynamic RPC Marshalling:
  N. Feske: *"A case study on the cost and benefit of dynamic RPC marshalling for low-level system components"*
- DSI:
  Reuther, Löser, Härtig: *"A Streaming Interface for Real Time Interprocess Communication"*
- Singularity IPC:
  Faehndrich, Aiken et al.: *"Language support for fast and reliable message-based communication in Singularity OS"*

Coming soon

- So far: Basic Abstractions
  - Tasks & Threads
  - Memory
  - IPC

- **Next weeks**: Getting larger – Building system components
  - Resource Management (Nov 11)
  - Device Drivers (Nov 18)
- **Exercise**:
  - Brinch-Hansen – The nucleus of a multiprogramming system