



Inter-Process Communication

Nils Asmussen

Dresden, 2025-10-21



- Microkernels as a design alternative
 - Flexibility
 - Security
- Case Study: Fiasco.OC
 - Provides: Tasks, Threads, Communication
 - Capabilities to denote kernel objects



- Inter-Process Communication (IPC)
 - Purpose
 - Implementation
 - How to find a service?
 - Tool/Language support
 - Security – Who speaks to whom?
 - Shared memory

- IPC is a fundamental mechanism in L4:
 - Exchange data
 - Synchronization
 - Sleep, timeout
 - Hardware / software interrupts
 - Grant access to resources (memory, I/O ports, kernel objects)
 - Exceptions
- Liedtke: *“IPC performance is the master.”*

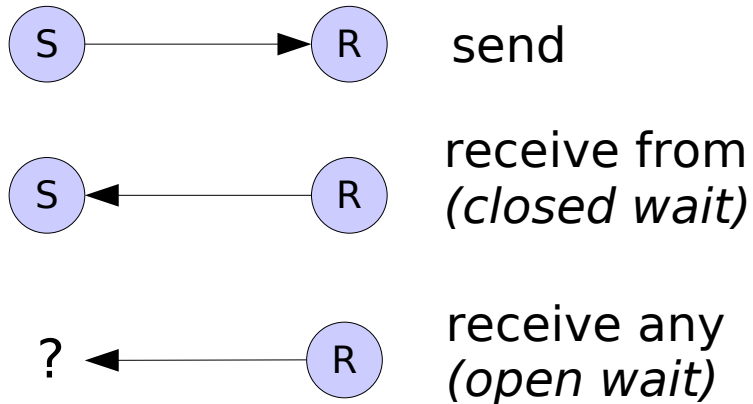
- Asynchronous IPC (e.g., Mach)
 - “Fire and forget”
 - In-kernel message buffering
 - Two problems:
 - Data copied twice
 - DoS attack on kernel memory (never receive data) – can use quotas, though
- Synchronous IPC (e.g., L4)
 - IPC partner blocks until other one gets ready
 - Direct copy between sender and receiver



- What you can send:
 - Plain data
 - Resource mappings
- Types
 - Send
 - Closed wait
 - Open wait
 - Call
 - Reply & wait

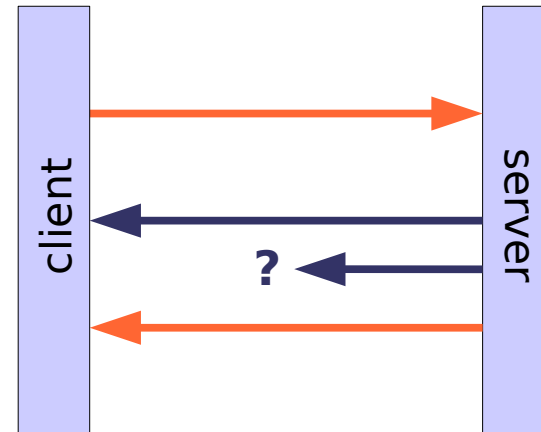
- Timeouts
 - 0 (non-blocking IPC)
 - NEVER or specific value – block until partner gets ready or timeout occurs
 - sleep() is implemented as IPC to NIL (non-existing) thread with timeout
- Exceptions
 - Certain conditions need external interaction
 - Page faults
 - L4Linux system calls
 - Virtualization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

Basics



- Why is there no broadcast?

Special cases for client/server IPC



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



Purpose

Implementation

Tool/Language support

Security

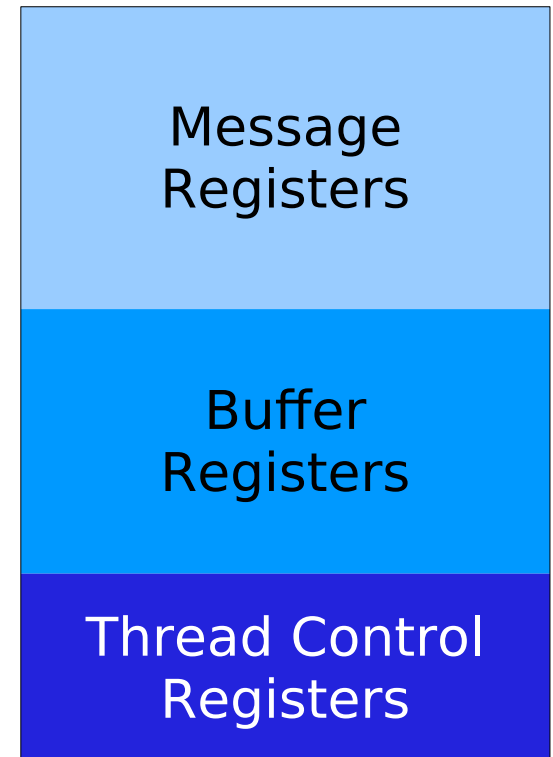
How to find a service?

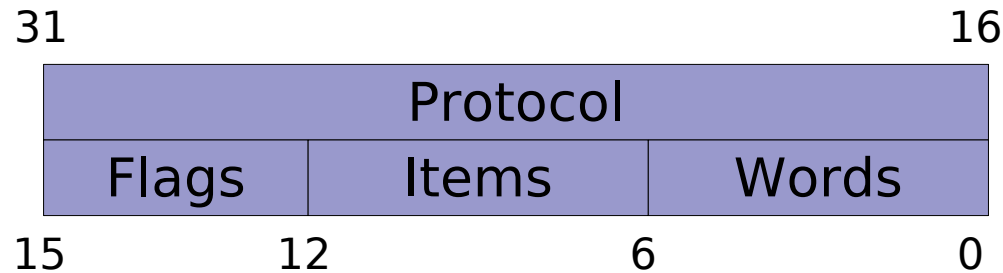
Shared memory

- Referenced through a capability (local name)
- Created using *factory* object
 - Each L4Re task is assigned a **factory object**
 - Factory creates other objects (e.g., kernel objects)
 - Can enforce quotas / perform accounting / ...
- Bound to a thread (receiver)
 - IPC channels are uni-directional
 - Anyone with the gate capability may send, only bound thread receives
- Add a label
 - A thread may receive from multiple gates
 - Label allows to identify where a message came from

- Sending:
 - Sender calls send or call
 - Blocks until message has been sent or reply has been received
- Receiving:
 - Receiver calls open wait.
 - Waits for message on any of its gates
 - Receive system call returns label of the used gate (but not the sender's capability!)
- Replying
 - Receiver doesn't have IPC Gate to sender.
 - Kernel provides implicit reply capability (per-thread)
 - Valid until reply sent or next wait started.

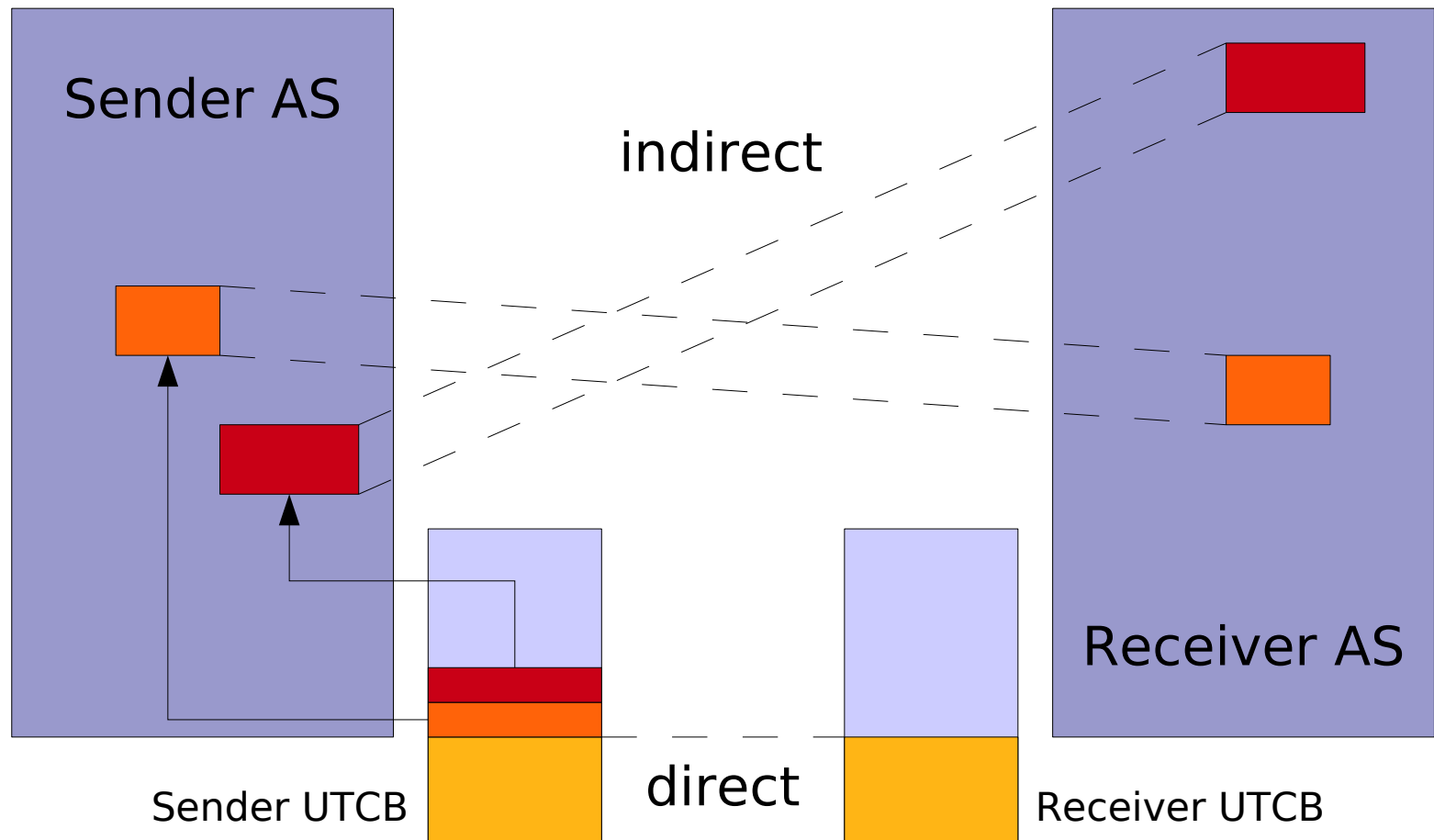
- **U**ser-level **T**hread **C**ontrol **B**lock
- Set of “virtual” registers
- Message Registers
 - System call parameters
 - IPC: direct copy to receiver
- Buffer registers
 - Contains flexpage descriptors
- Thread Control Registers
 - Thread-private data
 - Preserved, not copied





- **Protocol:**
 - User-defined type of communication
 - Pre-defined system protocols (Page fault, IRQ, ...)
- **Flags**
 - Special-purpose communication flags
- **Items**
 - Number of indirect items to copy (resource mappings)
- **Words**
 - Number of direct items to copy (plain data)

Direct vs. indirect copy





Purpose

Implementation

Tool/Language support

Security

How to find a service?

Shared memory

Client-Server RPC Broken down

Client

Marshall data
Assign Opcode
IPC call

Server

IPC wait
Unmarshall Opcode
Unmarshall Data
Execute function
Marshall reply or error
IPC reply
Goto begin

Unmarshall error or reply



```
/* Arguments: 1 integer parameter, 1 char array with size */
int FOO_OP1_call(l4_cap_idx_t dest, int arg1, char *arg2, unsigned size) {
    int idx = 0; // index into message registers

    // opcode and first arg go into first 2 registers
    l4_utcb_mr()->mr[idx++] = OP1_opcode;
    l4_utcb_mr()->mr[idx++] = arg1;

    // tricky: memcpy buffer into registers, adapt idx according
    //         to size (XXX NO BOUNDS CHECK!!!)
    memcpy(&l4_utcb_mr()->mr[idx], arg2, size);
    idx += round_up(size / sizeof(int));

    // create message tag (prototype, <idx> words, no bufs, no flags)
    l4_msgtag_t tag = l4_msg_tag(PROTO_FOO, idx, 0, 0);
    return l4_ipc_call(dest, l4_utcb(), tag, TIMEOUT_NEVER);
}
```

- Now repeat the above steps for
 - $N > 20$ functions with
 - varying parameters
 - varying argument size
 - complex use of send/receive flexpages
 - correct error checking
 - ...
- Dull and **error-prone!**

- Specify the interface of server in *Interface Definition Language* (IDL)

```
interface F00 {  
    int OP1(int arg1,  
            [size_is(arg2_size)] char *arg2,  
            unsigned arg2_size);  
};
```

- Use IDL Compiler to generate IPC code
 - Automatic assignment of RPC opcodes
 - Generated marshalling/unmarshalling code
 - Built-in error handling
 - Client/server stub functions to fill in
- For L4: Dice – **D**ROPS **I**DL **C**ompiler

- 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
 - ...

- C++: streams
- Overload operator<< to access the UTCB
 - Copying of basic data types and arrays into message registers
 - Dedicated objects representing flexpages copied into buffer registers
 - Automatic updates of positions in buffer
- Do the reverse steps for operator>>

```
int Foo::op1(l4_cap_idx_t dest, int arg1,  
            char *arg2, unsigned arg2_size)  
{  
    int res = -1;  
    L4_ipc_iostream i(l4_utcb());  
    i << Foo::Op1  
      << arg1  
      << Buffer(arg2, arg2_size);  
    int err = i.call(dest);  
    if (!err)  
        i >> res;  
    return res;  
}
```

```
int Foo::dispatch(L4_ipc_iostream& str, l4_msgtag_t tag) {  
    // check for invalid invocations  
    if (tag.label() != PROT0_F00)  
        return -L4_ENOSYS;  
  
    int opcode, arg1, retval;  
    Buffer argbuf(MAX_BUF_SIZE);  
  
    str >> opcode;  
    switch(opcode) {  
        case Foo::Op1:  
            str >> arg1 >> argbuf;  
            // do something clever, calculate retval  
            str << retval;  
            return L4_EOK;  
            // .. more cases ..  
    }  
}
```

- C++-based operating system framework
- Abstract from the underlying kernel
 - Runs on Linux, Fiasco.OC, OKL4, L4::Pistacchio, Nova, CodeZero
 - 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

- 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
(Prototype: Liasis IDL compiler by Stefan Kalkowski, 2008)



Purpose

Implementation

Tool/Language support

Security

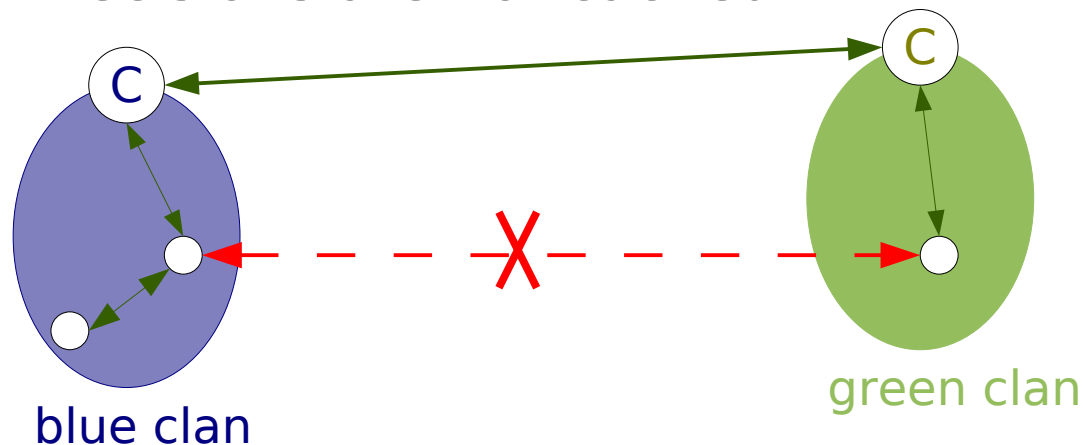
How to find a service?

Shared memory

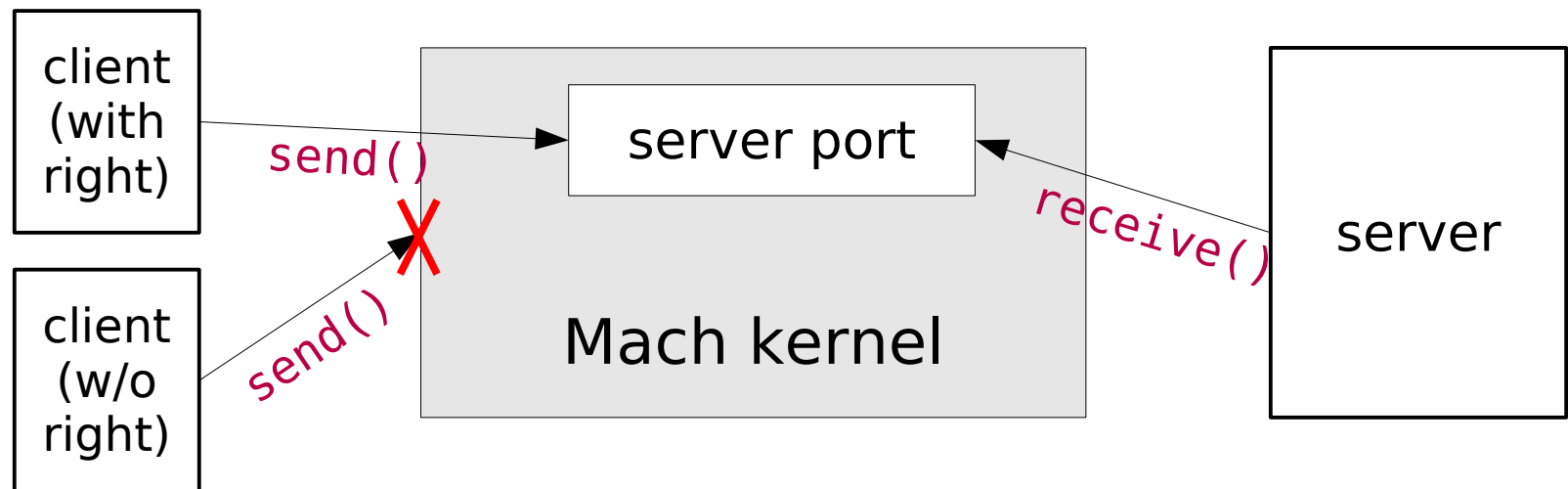


- Problem: How to control data flow?
- Crucial problem to solve when building real systems
- Many proposed solutions

- 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.

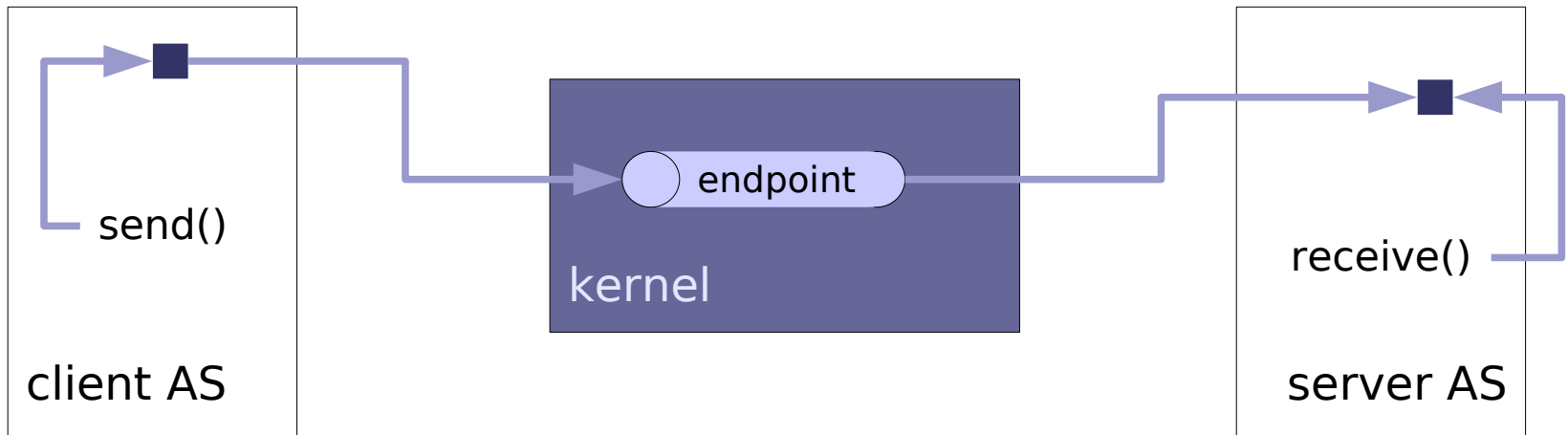


- Dedicated kernel objects (global)
- Applications hold send/recv rights for ports
- Kernel checks whether task owns sufficient rights before doing IPC



- 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 exception in the beginning

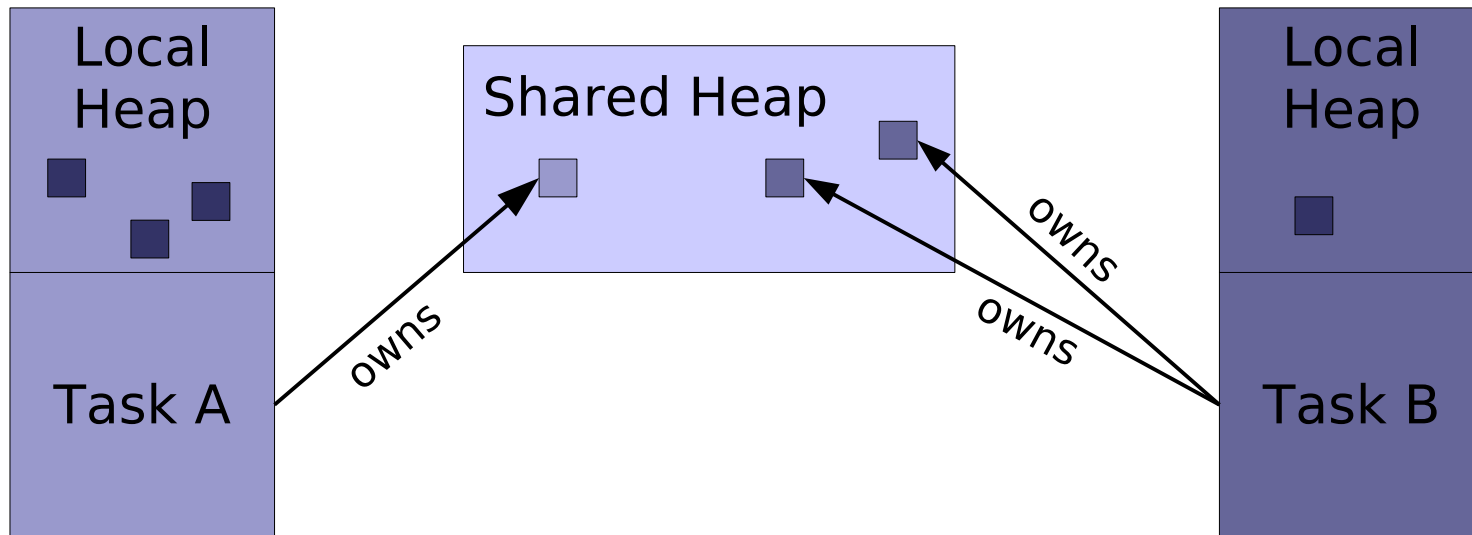
- Idea:
 - Invoke IPC on a kernel-object (IPC gate)
 - > **endpoint (capability)**
 - Kernel object mapped to a virtual address (local name space)
 - task only knows object's local name
 - no information leaks through global names



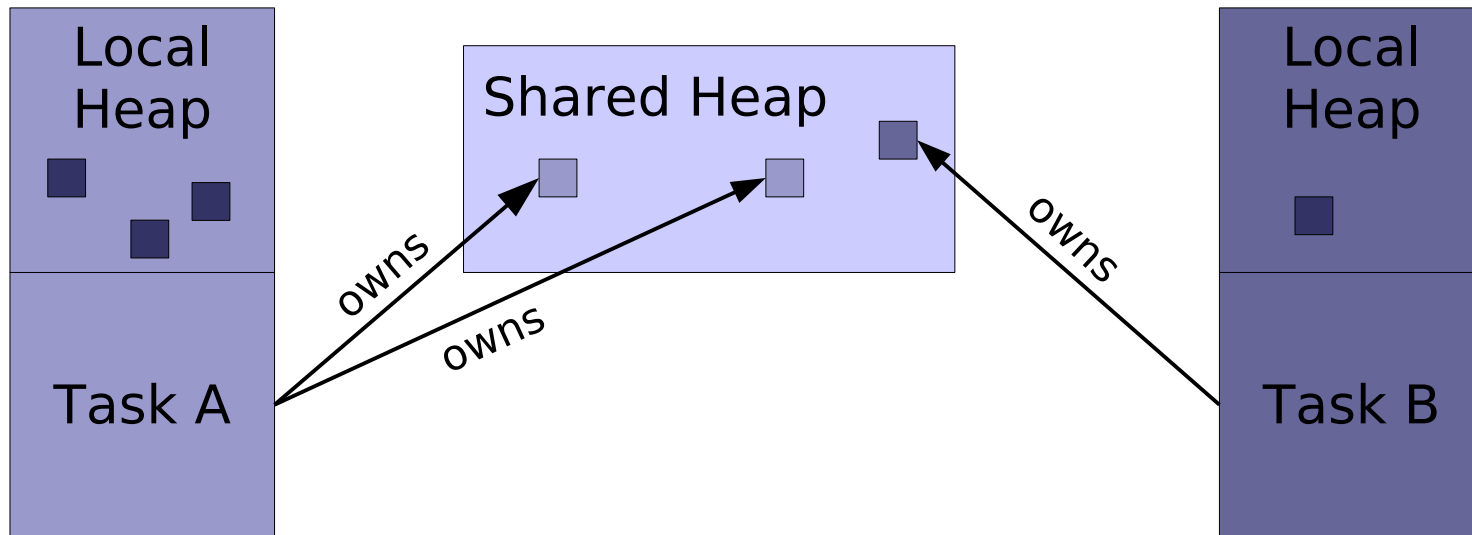


- Singularity
 - Research microkernel by MS Research
 - Written in a dialect of C# (Sing#)
 - Topic of a paper reading exercise
- All applications run in privileged mode.
 - No system call overhead – syscalls are normal function calls
- Enforce system safety at compile time.
 - Isolation completely realized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



- 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





- Mechanisms for controlling 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)



Purpose

Implementation

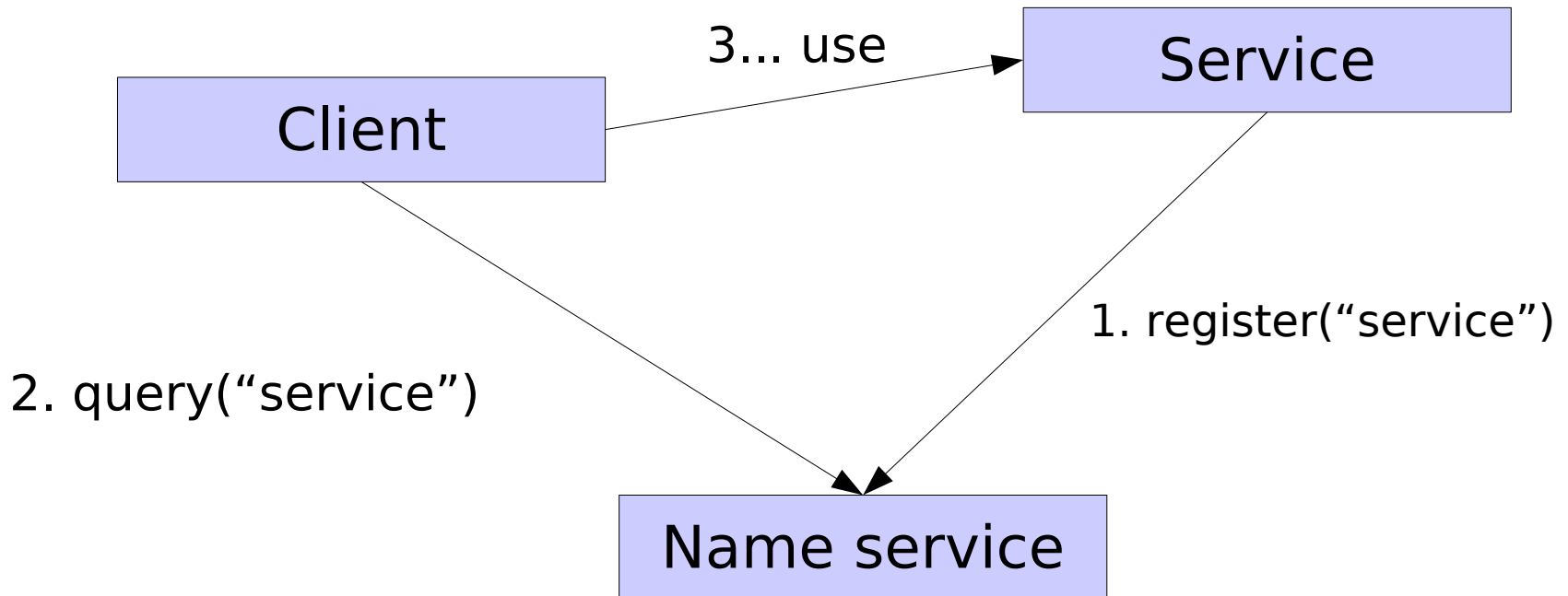
Tool/Language support

Security

How to find a service?

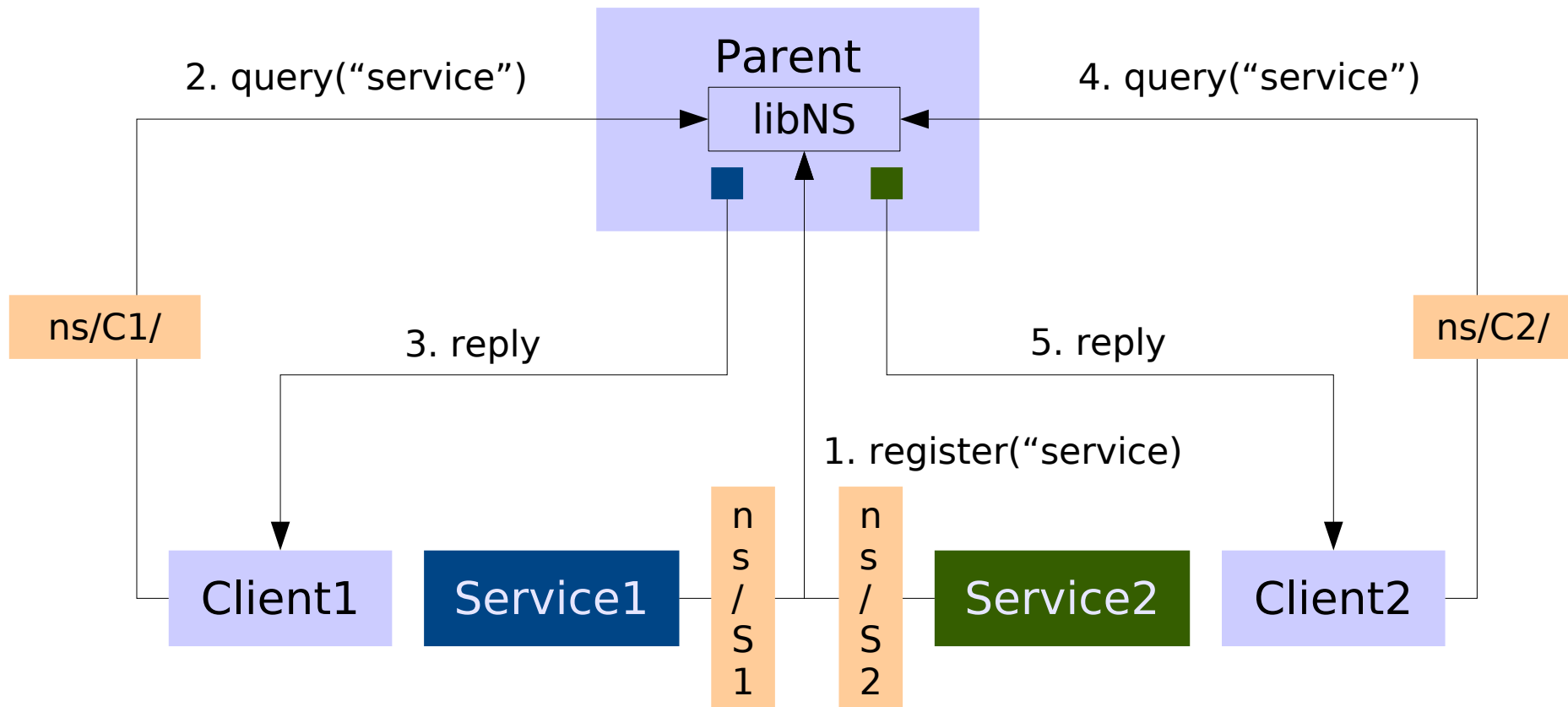
Shared memory

- Need to get some kind of identification of service provider in order to perform IPC.
 - L4Re: need to get a capability mapped into my local capability space
- Idea borrowed from the internet: translate human-readable-names into IDs.
- Need a name service provider.



- **Race condition:** Evil app can register name before real one.
- **Information leak:** Query name service for names and gain information about running services → contradicts resource separation
 - *Names are a resource and must be managed!*

Hierarchical naming



- Race Condition
 - Parent controls name space and program startup
 - Knows who is registering a service
- Information leak
 - Parent only provides name space content to each application
- Problem: configuration can be a mess.



Purpose

Implementation

Tool/Language support

Security

How to find a service?

Shared memory

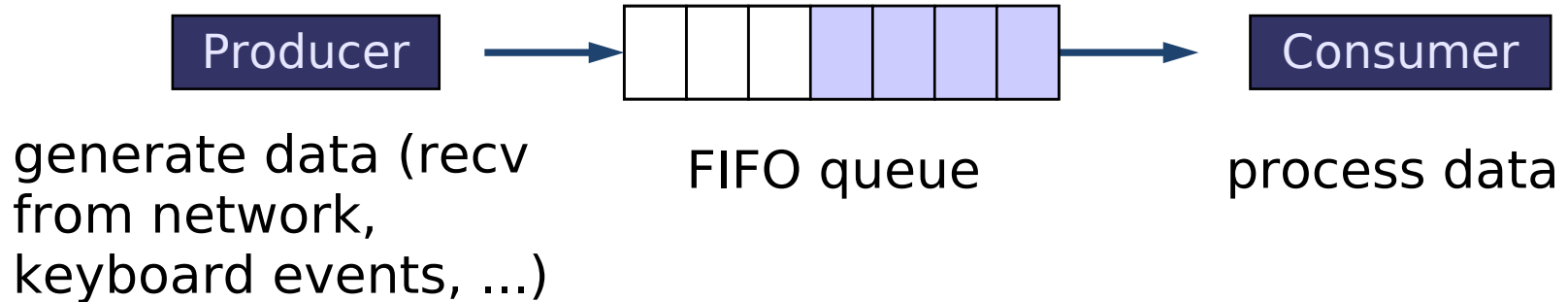
- Some applications need high throughput for a lot of data.
 - Sharing memory between tasks can provide better performance
- Many workloads need asynchronous communication.
 - Fiasco.OC: IRQ kernel object



- 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

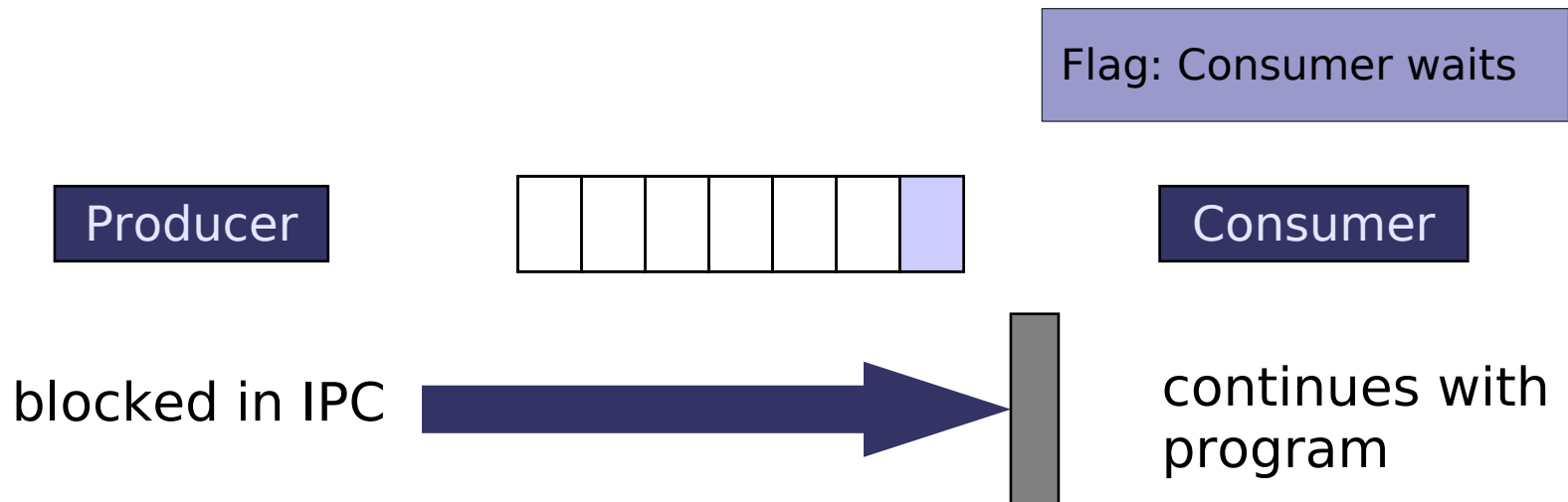
Example: Consumer-Producer Problem

- 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



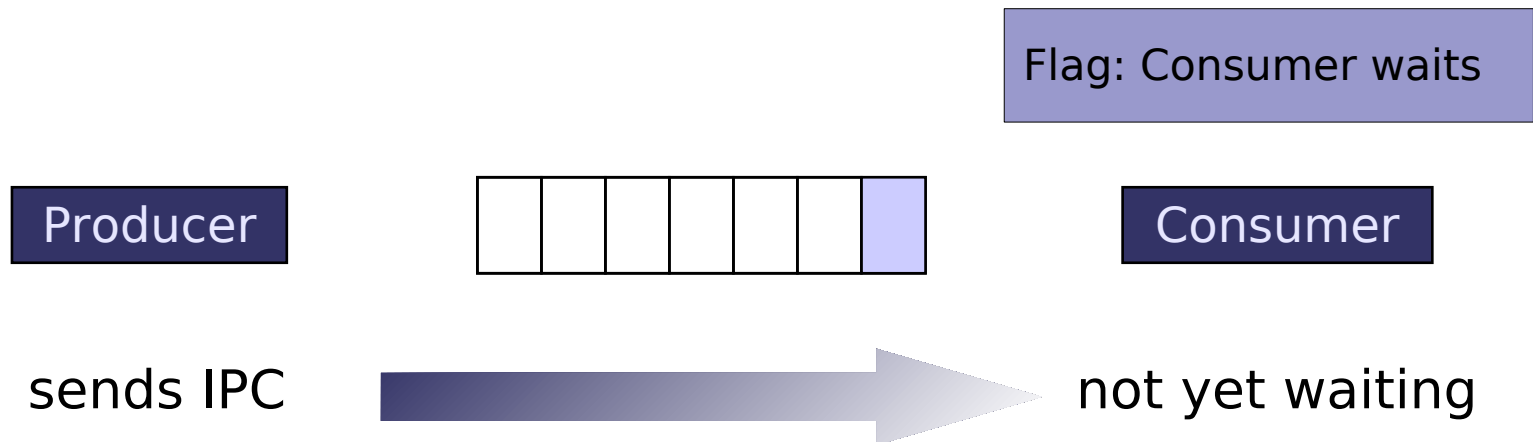
1st try: Consumer sets flag

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



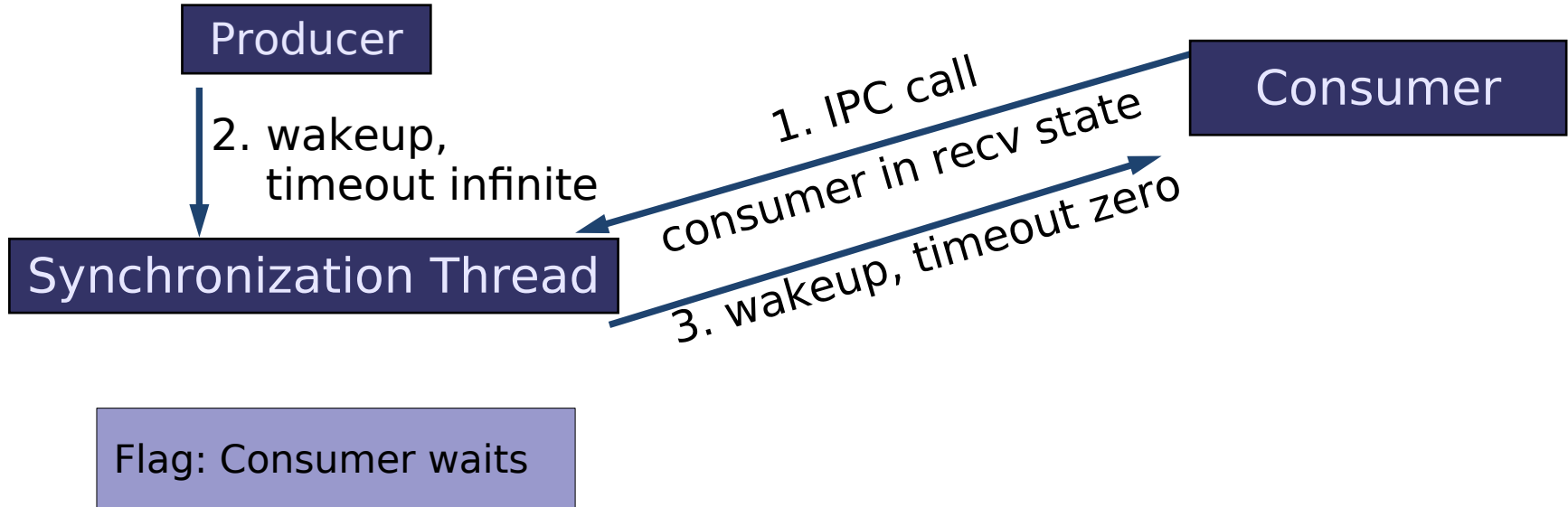
2nd try: Notify with zero Timeout

- 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

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



- L4 kernel manual: <http://l4hq.org/docs/manuals/Ln-86-21.pdf>
- Genode Dynamic RPC Marshalling:
N. Feske: *“A case study on the cost and benefit of dynamic RPC marshalling for low-level system components”*
- Singularity IPC:
Faehndrich, Aiken et al.: *“Language support for fast and reliable message-based communication in Singularity OS”*