Managarm: A Fully Asynchronous Operating System

Alexander van der Grinten
Kacper Słomiński
Geert Custers

The Managarm Project

FG-BS Frühjahrstreffen, March 14 - 15, 2024

What is Managarm?

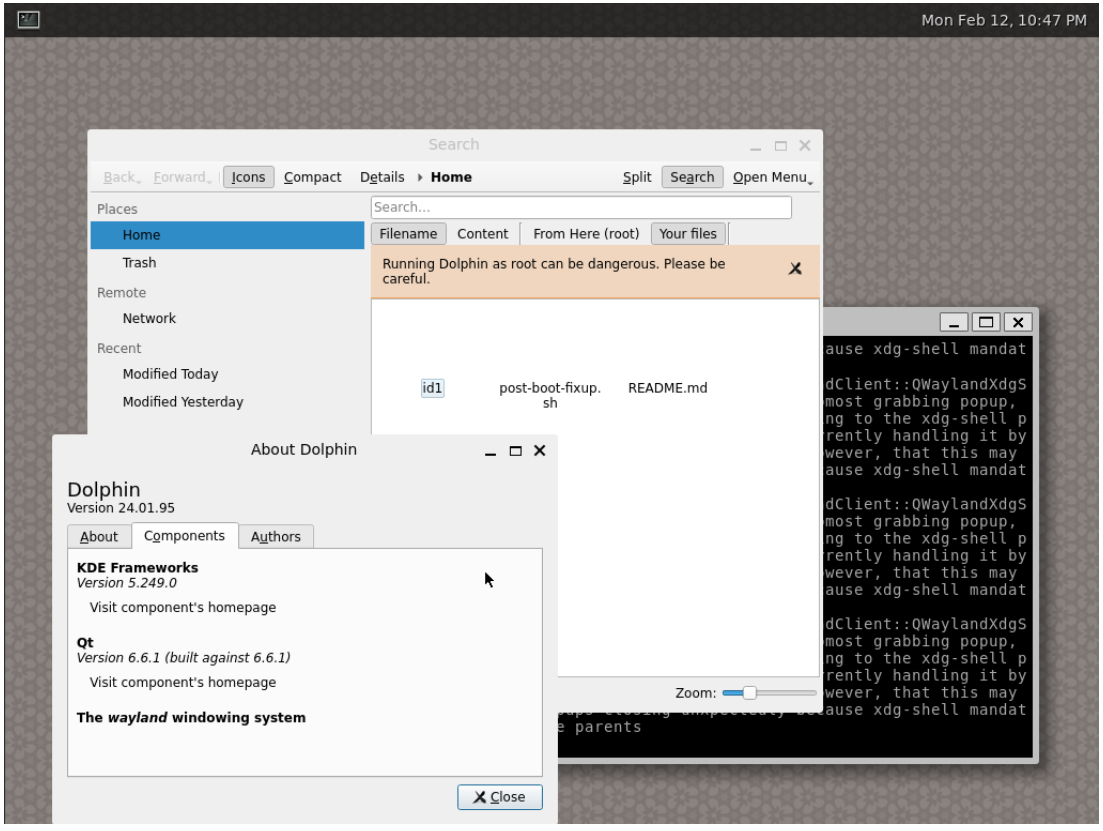
Managarm: microkernel-based **general-purpose** OS with focus on asynchronous I/O.

What is Managarm?

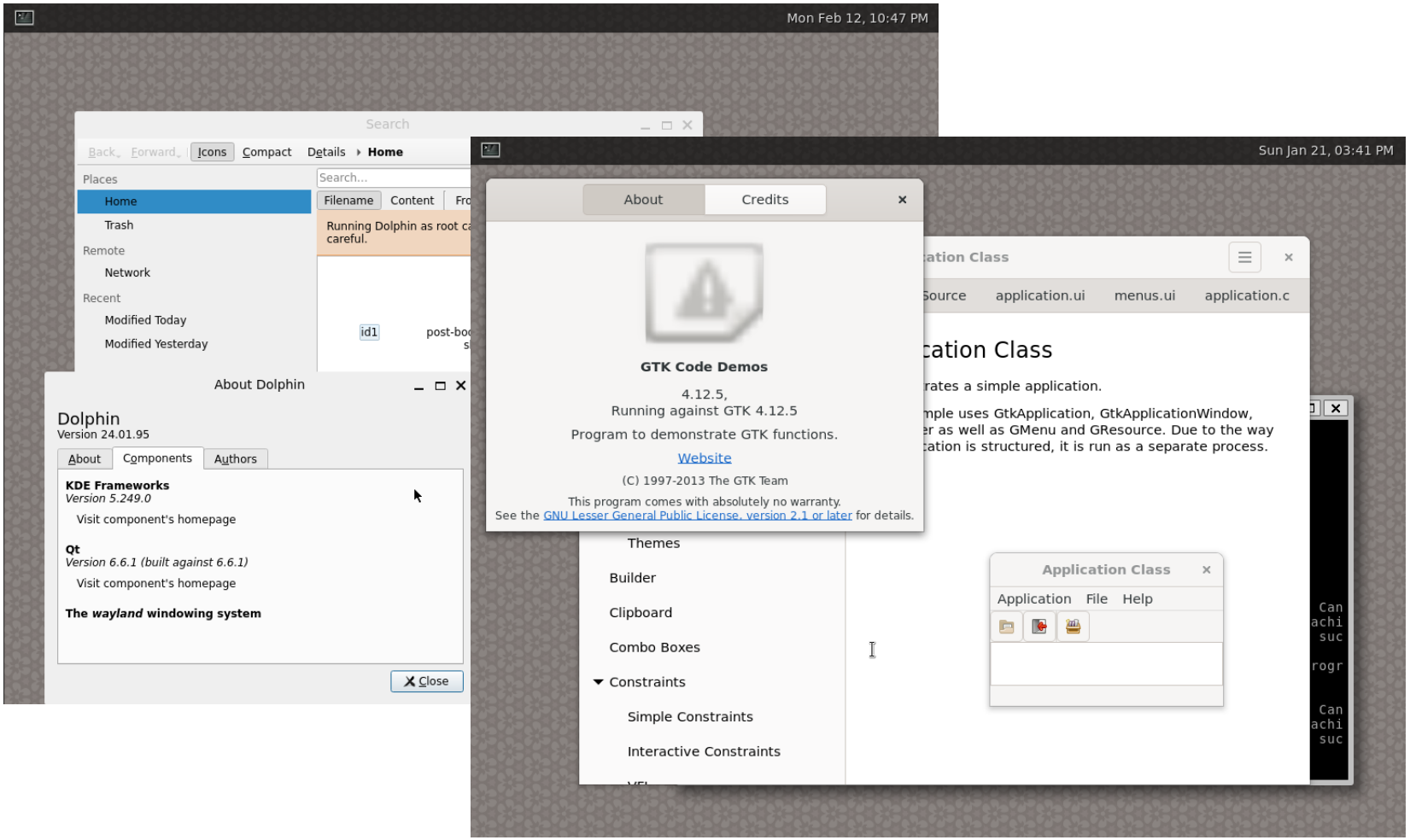
Managarm: microkernel-based **general-purpose** OS with focus on asynchronous I/O.

- Open-source project, mostly written in modern C++.
- Started in 2014.
- Many active contributors.

What is Managarm?



What is Managarm?



What is Managarm?

- Fully asynchronous = (almost) all system calls are asynchronous (same for drivers/servers, ...).

What is Managarm?

- Fully asynchronous = (almost) all system calls are asynchronous (same for drivers/servers, ...).
- Reasonably good source-level compatiblity to Linux via user space emulation.

What is Managarm?

- Fully asynchronous = (almost) all system calls are asynchronous (same for drivers/servers, ...).
- Reasonably good source-level compatiblity to Linux via user space emulation.

SotA L4 style kernels: focus on low latency synchronous IPC

What is Managarm?

- Fully asynchronous = (almost) all system calls are asynchronous (same for drivers/servers, ...).
- Reasonably good source-level compatiblity to Linux via user space emulation.

SotA L4 style kernels: focus on low latency synchronous IPC

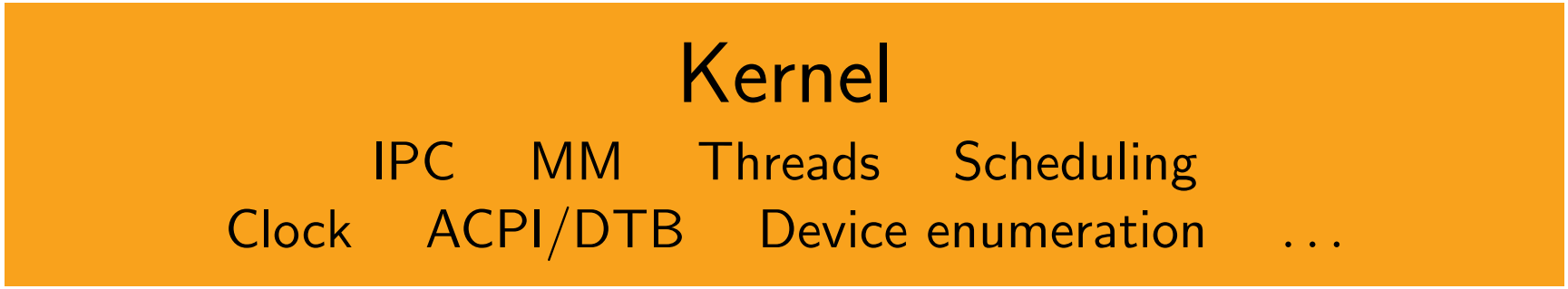
Advantage of asynchronicity: can handle higher bandwidth of concurrent requests and uses fewer resources than synchronous approaches
(e.g., fewer threads, less RAM).

Agenda

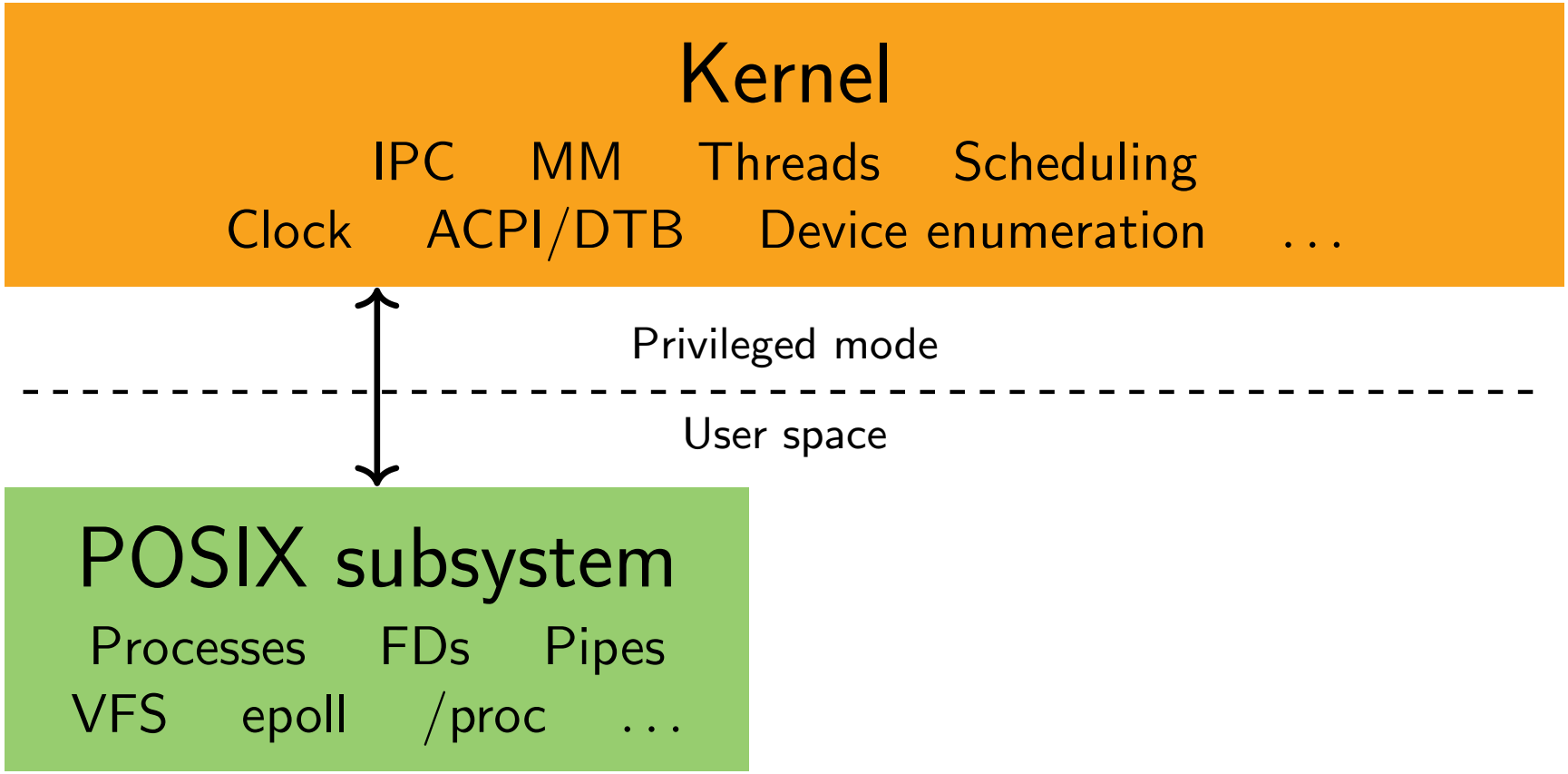
- System Architecture
- Inter-Process Communication (IPC)

System Architecture

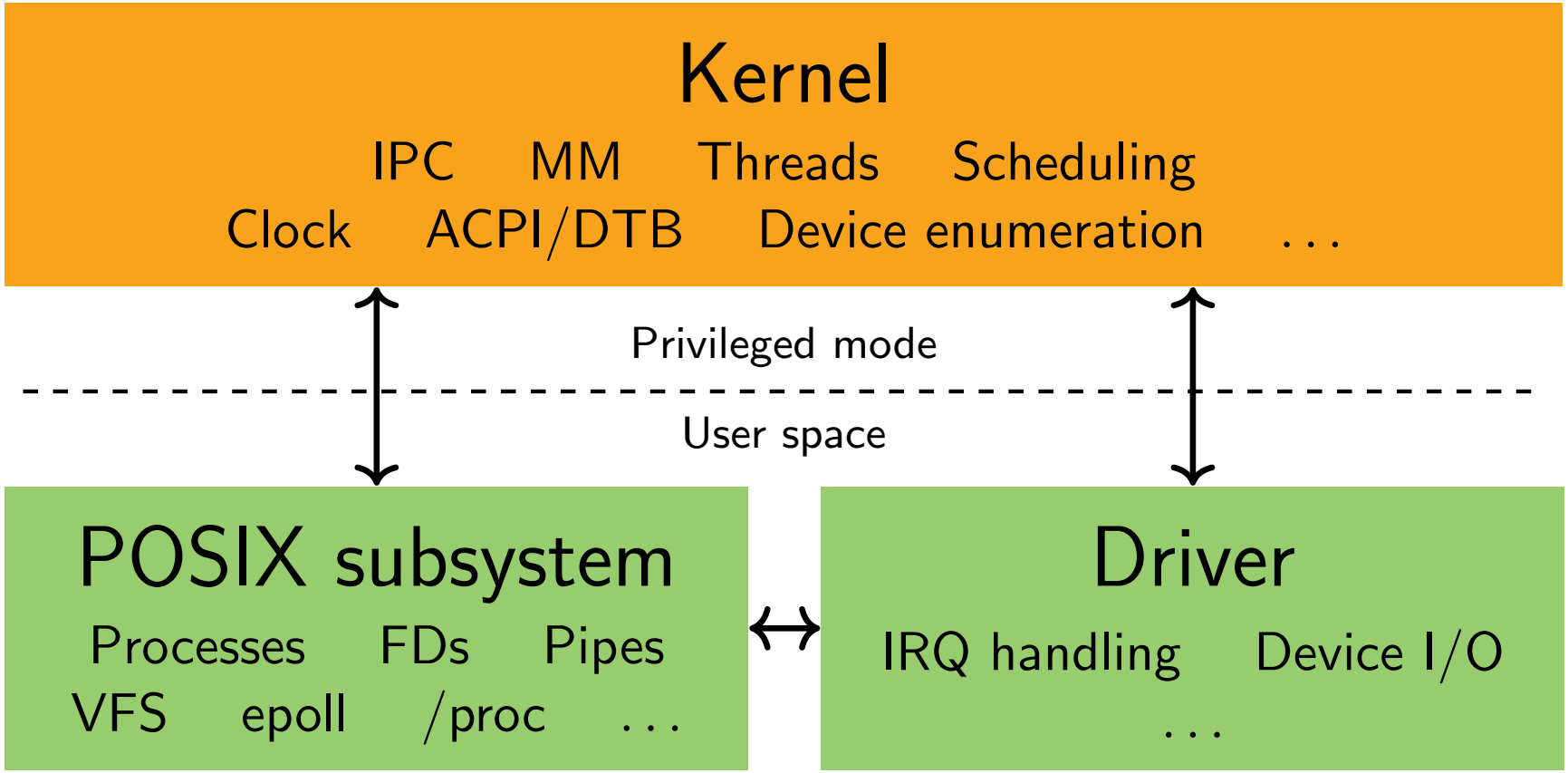
Managarm: block diagram



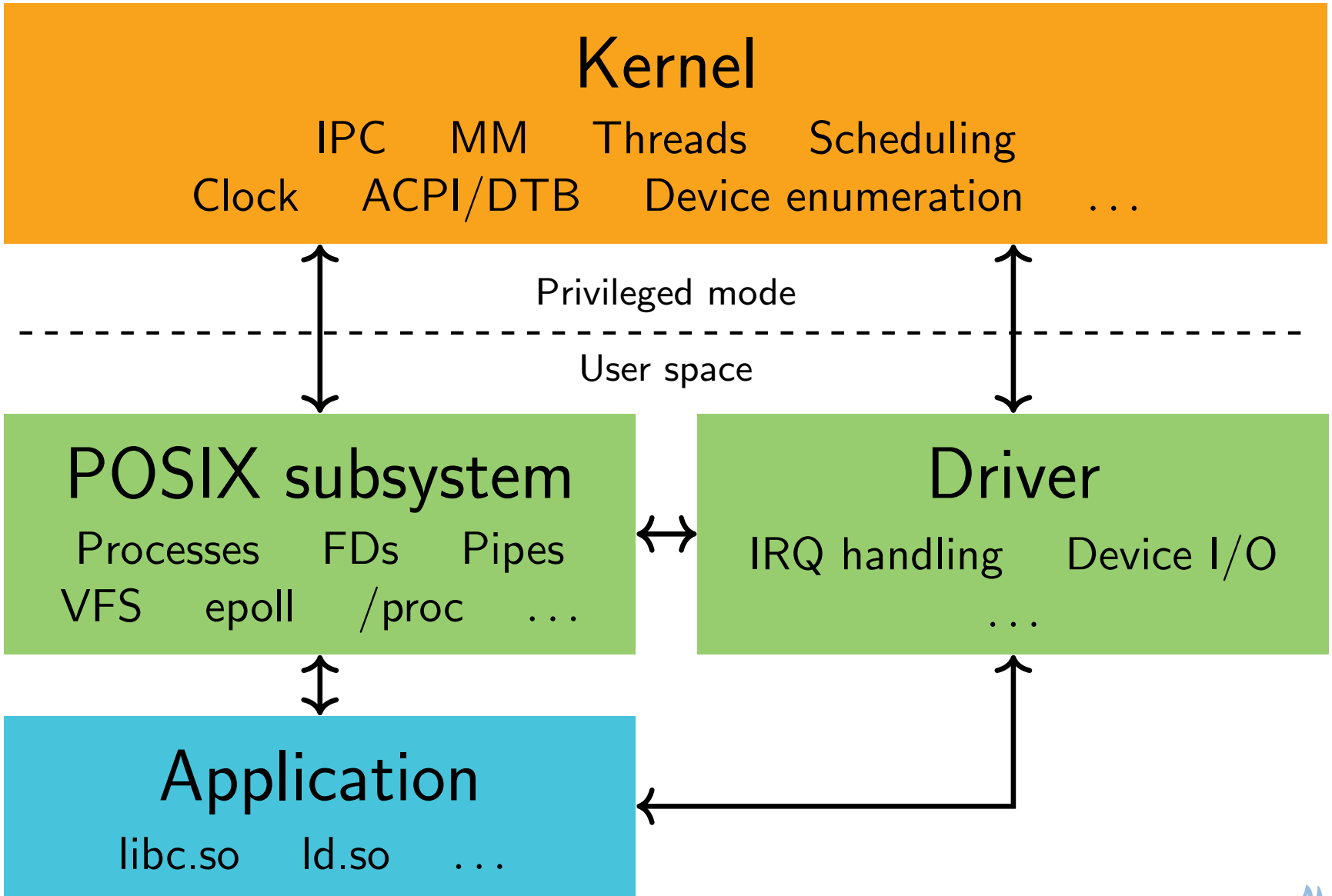
Managarm: block diagram



Managarm: block diagram



Managarm: block diagram



Kernel $\leftrightarrow$ user space interaction

In contrast to synchronous syscalls: async syscalls do not complete when control flow returns to user space.

Kernel $\leftrightarrow$ user space interaction

In contrast to synchronous syscalls: async syscalls do not complete when control flow returns to user space.

Instead: **lock-free ring buffer** is used to notify user space whenever an async syscall completes.

Kernel $\leftrightarrow$ user space interaction

In contrast to synchronous syscalls: async syscalls do not complete when control flow returns to user space.

Instead: **lock-free ring buffer** is used to notify user space whenever an async syscall completes.

Use C++20 coroutines to write ergonomic asynchronous code:
`co_await some_async_operation()`

Kernel $\leftrightarrow$ user space interaction

In contrast to synchronous syscalls: async syscalls do not complete when control flow returns to user space.

Instead: **lock-free ring buffer** is used to notify user space whenever an async syscall completes.

Use C++20 coroutines to write ergonomic asynchronous code:
`co_await some_async_operation()`

- Reduces effort to write async code to level that is similar to synchronous code.

Kernel $\leftrightarrow$ user space interaction

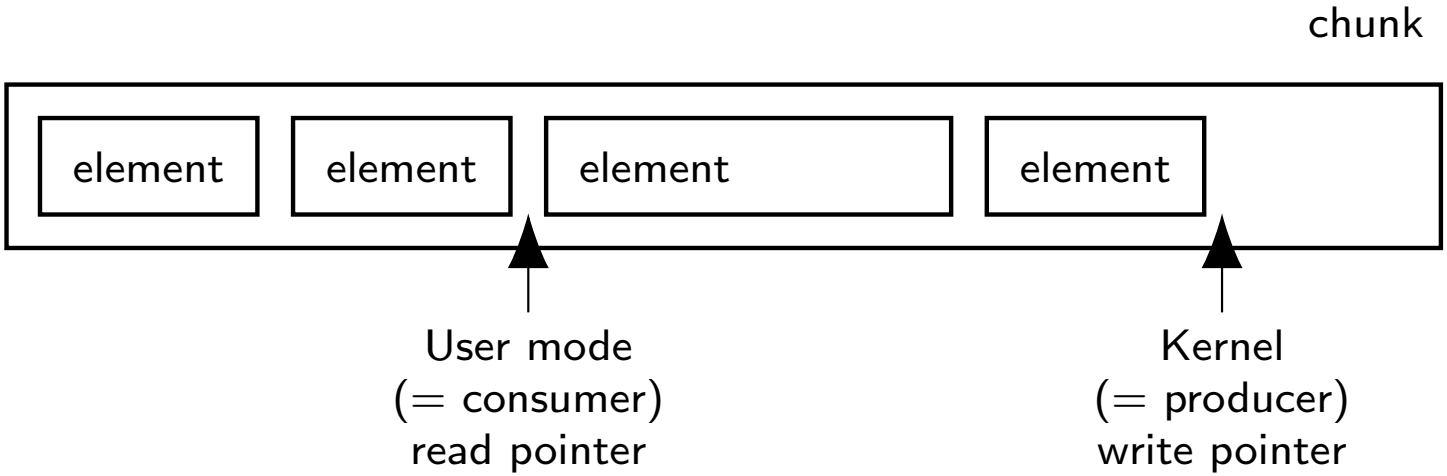
In contrast to synchronous syscalls: async syscalls do not complete when control flow returns to user space.

Instead: **lock-free ring buffer** is used to notify user space whenever an async syscall completes.

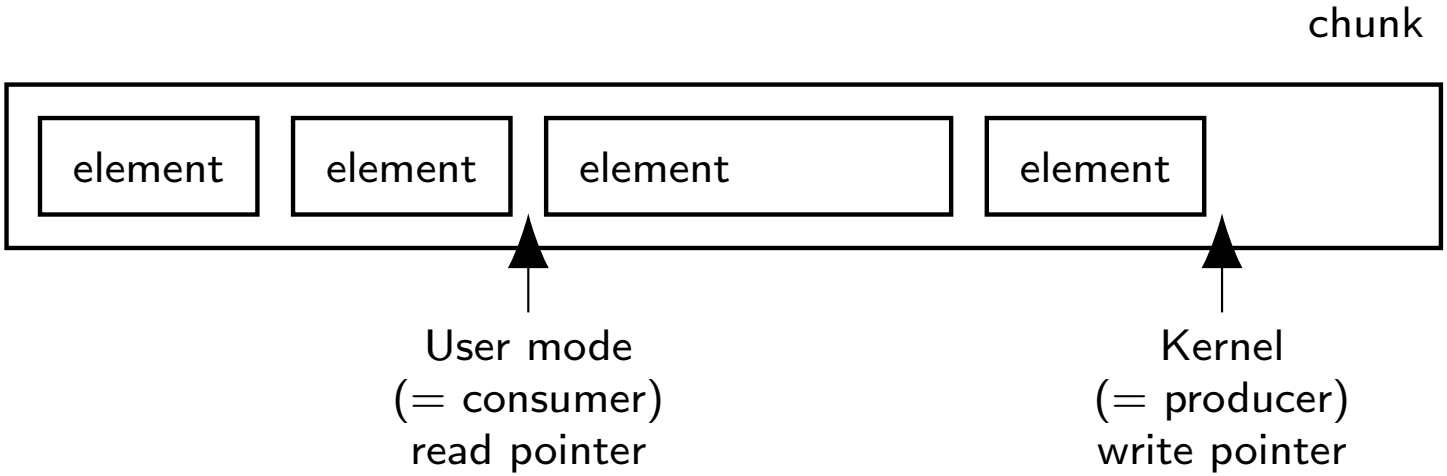
Use C++20 coroutines to write ergonomic asynchronous code:
`co_await some_async_operation()`

- Reduces effort to write async code to level that is similar to synchronous code.
- Other async/await mechanisms could be used as well (e.g., in Rust, Python, ...).

Notification ring buffer

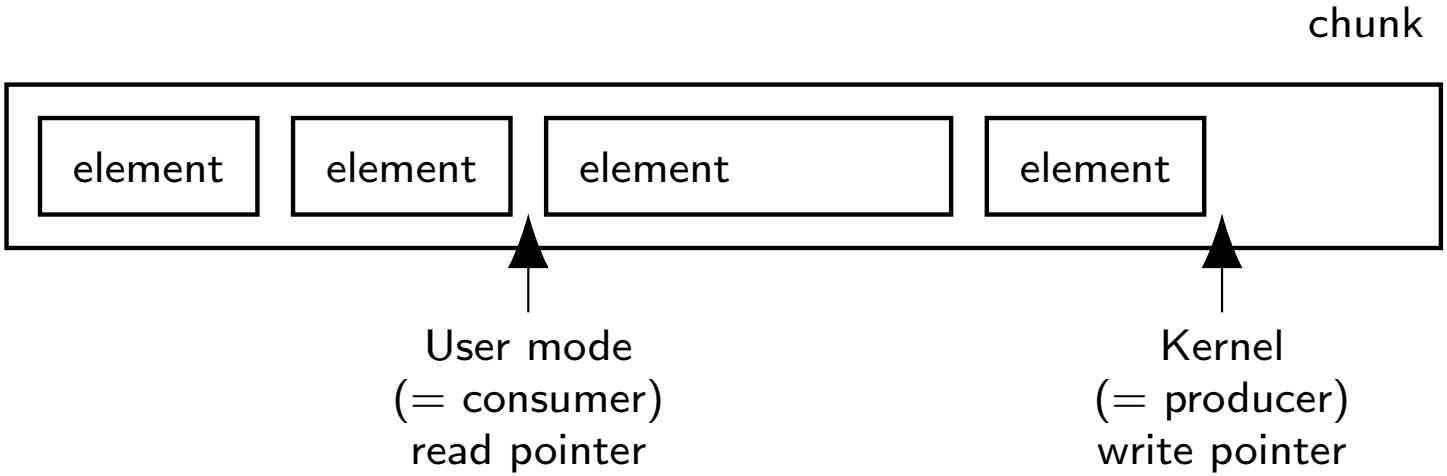


Notification ring buffer



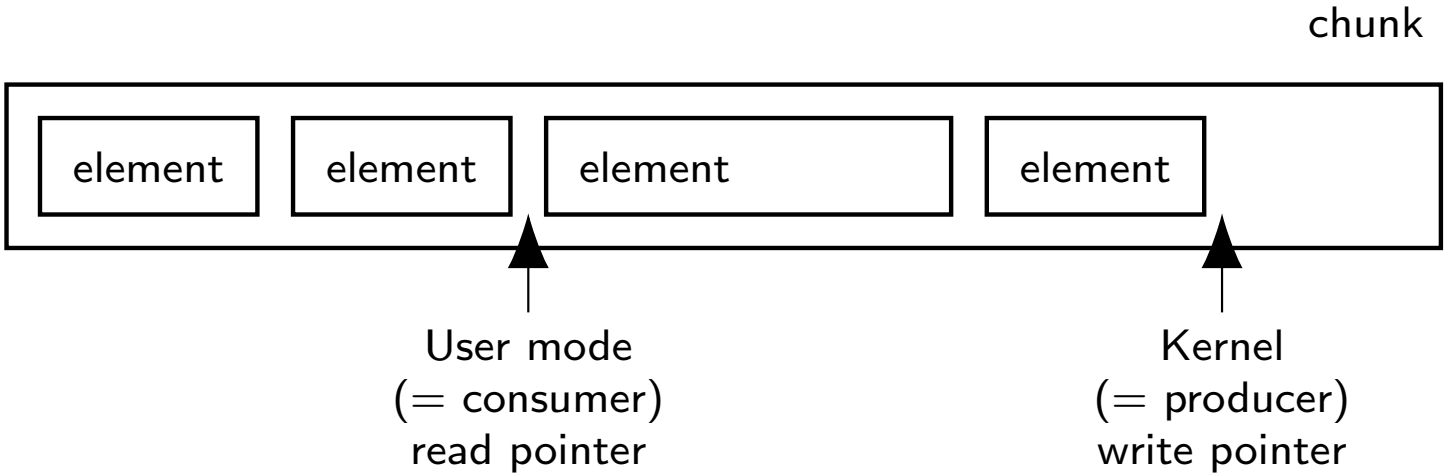
- Similar to io_uring in Linux, ...

Notification ring buffer



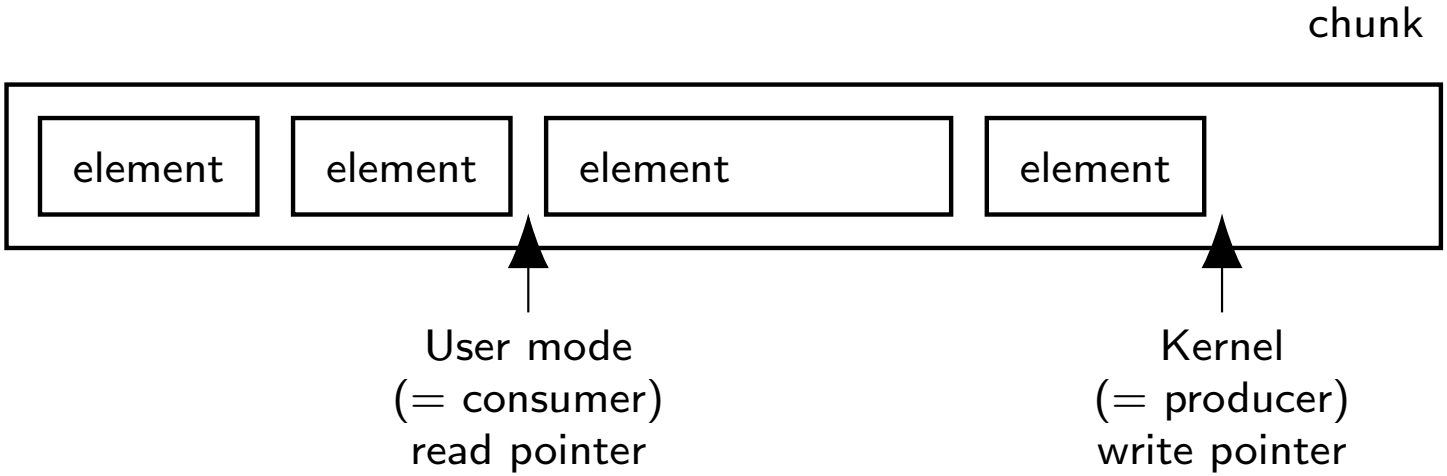
- Similar to io_uring in Linux, ...
- Retrieving notification requires **zero syscalls** on the fast path.

Notification ring buffer



- Similar to io_uring in Linux, ...
- Retrieving notification requires **zero syscalls** on the fast path.
  Syscall is only required if thread needs to block when no notifications are available.

Notification ring buffer



- Similar to io_uring in Linux, ...
- Retrieving notification requires **zero syscalls** on the fast path.
  Syscall is only required if thread needs to block when no notifications are available.
- User space uses a pointer-sized value to match completion notifications to pending syscalls.

Asynchronous system calls

Almost all syscalls are **async** in Managarm.

- Examples: IPC, mapping/unmapping memory, waiting for an IRQ, ...

Asynchronous system calls

Almost all syscalls are **async** in Managarm.

- Examples: IPC, mapping/unmapping memory, waiting for an IRQ, ...

Exception: system calls that explicitly synchronize threads; these are mostly **futex** operations.

Asynchronous system calls

Almost all syscalls are **async** in Managarm.

- Examples: IPC, mapping/unmapping memory, waiting for an IRQ, ...

Exception: system calls that explicitly synchronize threads; these are mostly **futex** operations.

- Required to implement mutexes, condition variables etc. in user space.

Asynchronous system calls

Almost all syscalls are **async** in Managarm.

- Examples: IPC, mapping/unmapping memory, waiting for an IRQ, ...

Exception: system calls that explicitly synchronize threads; these are mostly **futex** operations.

- Required to implement mutexes, condition variables etc. in user space.
- Also **required to block a thread** when there is no work to do.

Inter-Process Communication (IPC)

IPC: overview

- IPC is only dispatched when **both** sender/receiver are ready.

IPC: overview

- IPC is only dispatched when **both** sender/receiver are ready.
- IPC operations (e.g., send/receive) are queued.

IPC: overview

- IPC is only dispatched when **both** sender/receiver are ready.
- IPC operations (e.g., send/receive) are queued.
- Message contents (= bytes) are **not** queued.

IPC: overview

- IPC is only dispatched when **both** sender/receiver are ready.
- IPC operations (e.g., send/receive) are queued.
- Message contents (= bytes) are **not** queued.

Advantage: can handle efficiently arbitrary number of concurrent requests from a single thread.

IPC: overview

- IPC is only dispatched when **both** sender/receiver are ready.
- IPC operations (e.g., send/receive) are queued.
- Message contents (= bytes) are **not** queued.

Advantage: can handle efficiently arbitrary number of concurrent requests from a single thread.

Disadvantage: queuing of IPC operations requires memory allocations and bookkeeping within the kernel.

But: in-kernel representation of IPC operations is fixed-size and small.

IPC: overview

- IPC is only dispatched when **both** sender/receiver are ready.
- IPC operations (e.g., send/receive) are queued.
- Message contents (= bytes) are **not** queued.

Advantage: can handle efficiently arbitrary number of concurrent requests from a single thread.

Disadvantage: queuing of IPC operations requires memory allocations and bookkeeping within the kernel.

But: in-kernel representation of IPC operations is fixed-size and small.

$\rightsquigarrow$ Many cases achieve **competitive** performance with simpler synchronous IPC (e.g., in Linux)

IPC streams

Managarm uses "**streams**" as IPC primitive.

- Two endpoints per stream.

IPC streams

Managarm uses "**streams**" as IPC primitive.

- Two endpoints per stream.

Threads post "**actions**" (= IPC operations) to each endpoint of the stream.

- Examples: sending/receiving bytes, transfering capabilities.

IPC streams

Managarm uses "**streams**" as IPC primitive.

- Two endpoints per stream.

Threads post "**actions**" (= IPC operations) to each endpoint of the stream.

- Examples: sending/receiving bytes, transfering capabilities.
- Actions on both ends are **matched** against each other and dispatched: first action on first endpoint is matched to first action on second endpoint, ...

IPC streams

Managarm uses "**streams**" as IPC primitive.

- Two endpoints per stream.

Threads post "**actions**" (= IPC operations) to each endpoint of the stream.

- Examples: sending/receiving bytes, transfering capabilities.
- Actions on both ends are **matched** against each other and dispatched: first action on first endpoint is matched to first action on second endpoint, ...
- Actions on both ends of the stream must be **compatible**: send/receive ✓, send/send ✗.

IPC streams

Managarm uses "**streams**" as IPC primitive.

- Two endpoints per stream.

Threads post "**actions**" (= IPC operations) to each endpoint of the stream.

- Examples: sending/receiving bytes, transfering capabilities.
- Actions on both ends are **matched** against each other and dispatched: first action on first endpoint is matched to first action on second endpoint, ...
- Actions on both ends of the stream must be **compatible**: send/receive ✓, send/send ✗.

A single IPC syscall can submit multiple actions to a stream.

Data transfer

Basic actions to **send/receive bytes**: `SendFromBuffer` / `RecvToBuffer` (size must be known in advance).

Data transfer

Basic actions to **send/receive bytes**: `SendFromBuffer` / `RecvToBuffer` (size must be known in advance).

Other send/receive actions to avoid copying in certain situations:

- `RecvInline`: receive data to the notification ring buffer (as part of completion notification; bounded size).

Data transfer

Basic actions to **send/receive bytes**: `SendFromBuffer` / `RecvToBuffer` (size must be known in advance).

Other send/receive actions to avoid copying in certain situations:

- `RecvInline`: receive data to the notification ring buffer (as part of completion notification; bounded size).
- `SendFromBufferSg`: scatter-gather.
- ...

Data transfer

Basic actions to **send/receive bytes**: `SendFromBuffer` / `RecvToBuffer` (size must be known in advance).

Other send/receive actions to avoid copying in certain situations:

- `RecvInline`: receive data to the notification ring buffer (as part of completion notification; bounded size).
- `SendFromBufferSg`: scatter-gather.
- ...

All send* actions are compatible with all recv* actions.

Data transfer

Basic actions to **send/receive bytes**: `SendFromBuffer` / `RecvToBuffer` (size must be known in advance).

Other send/receive actions to avoid copying in certain situations:

- `RecvInline`: receive data to the notification ring buffer (as part of completion notification; bounded size).
- `SendFromBufferSg`: scatter-gather.
- ...

All send* actions are compatible with all recv* actions.

Other actions allow **transfering capabilities** or have specialized purposes (e.g., proving the identity of the thread that operates on the first endpoint to the second endpoint).

Request/response logic

It is often desirable to multiplex multiple concurrent requests/responses over a single stream.

Request/response logic

It is often desirable to multiplex multiple concurrent requests/responses over a single stream.

$\rightsquigarrow$ Allows multiple clients to talk to same server.

Request/response logic

It is often desirable to multiplex multiple concurrent requests/responses over a single stream.

$\rightsquigarrow$ Allows multiple clients to talk to same server.

Managarm uses "`Offer`"/"`Accept`" actions for this purpose.

- `Offer`/`Accept` pair creates a new "**ancillary**" stream
  (Ancillary stream is usually short-lived and discarded after a single request/response).

Request/response logic

It is often desirable to multiplex multiple concurrent requests/responses over a single stream.

$\rightsquigarrow$ Allows multiple clients to talk to same server.

Managarm uses "`Offer`"/"`Accept`" actions for this purpose.

- `Offer`/`Accept` pair creates a new "**ancillary**" stream (Ancillary stream is usually short-lived and discarded after a single request/response).
- Subsequent actions can be directed to the new stream (without the need to invoke an additional syscall).

Example: client/server IPC

1st endpoint,
client

2nd endpoint,
server

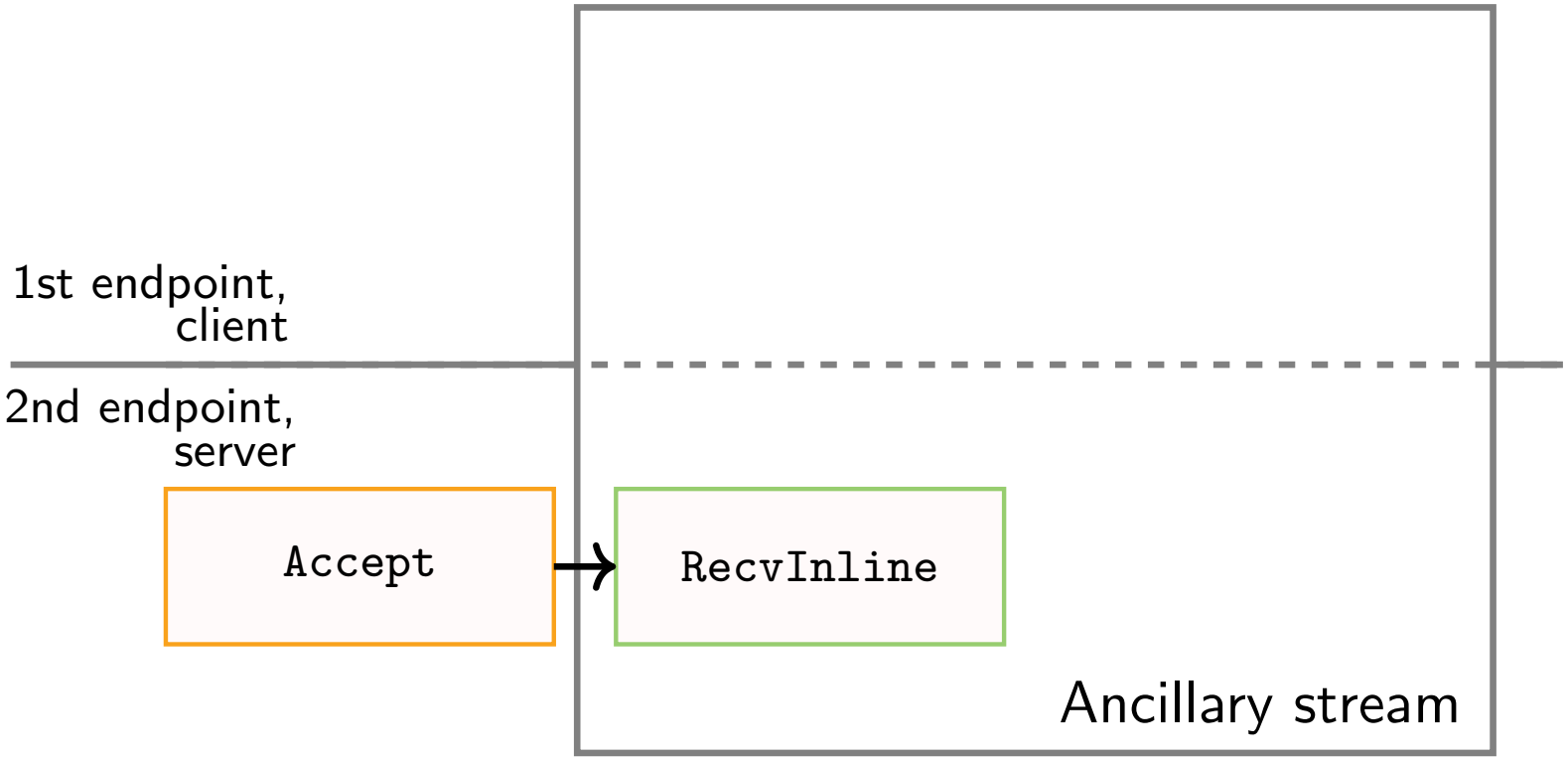
Example: client/server IPC

1st endpoint,
client

2nd endpoint,
server

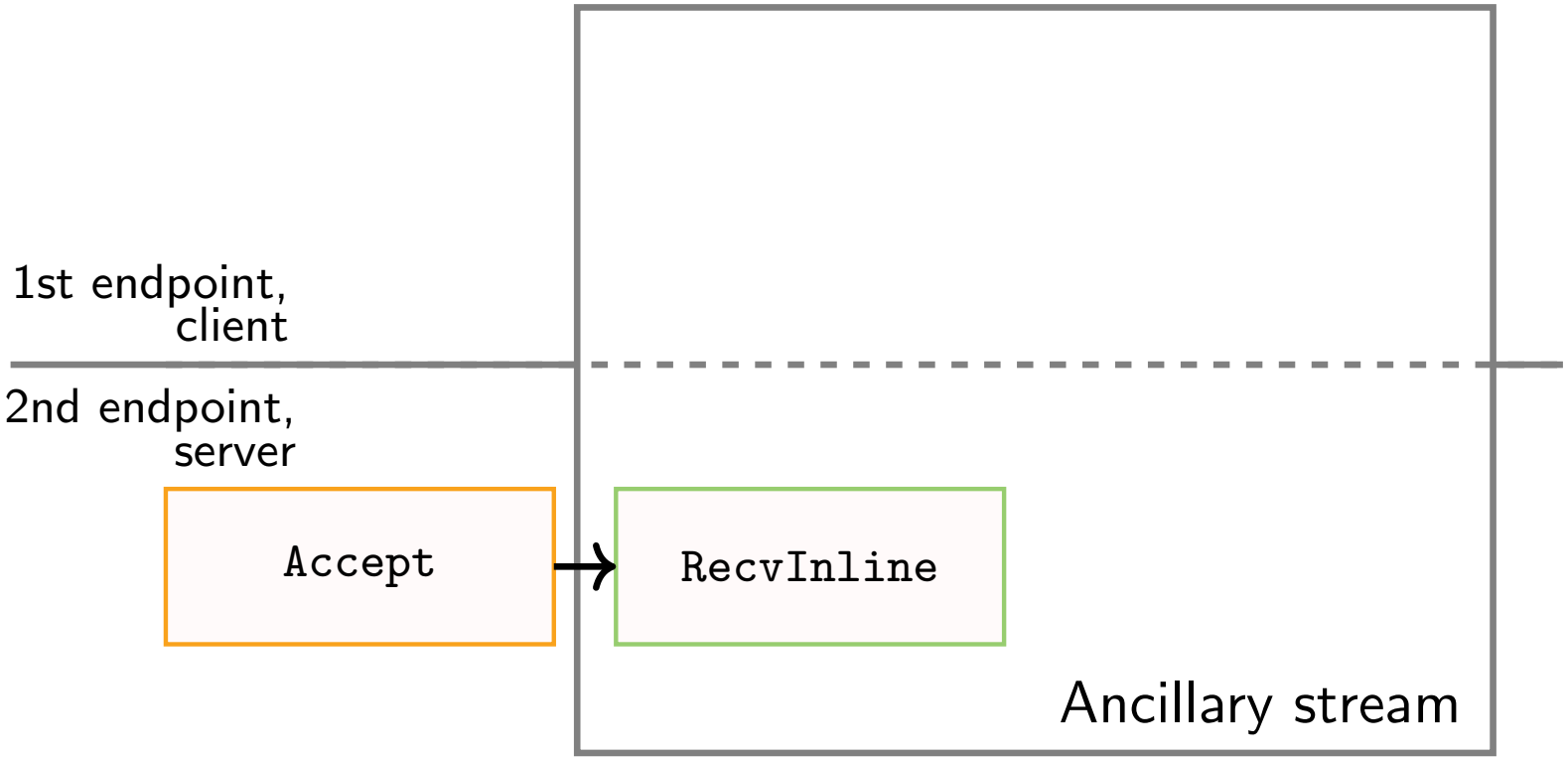
1. Server posts `Accept` → `RecvInline` (to receive a request).

Example: client/server IPC



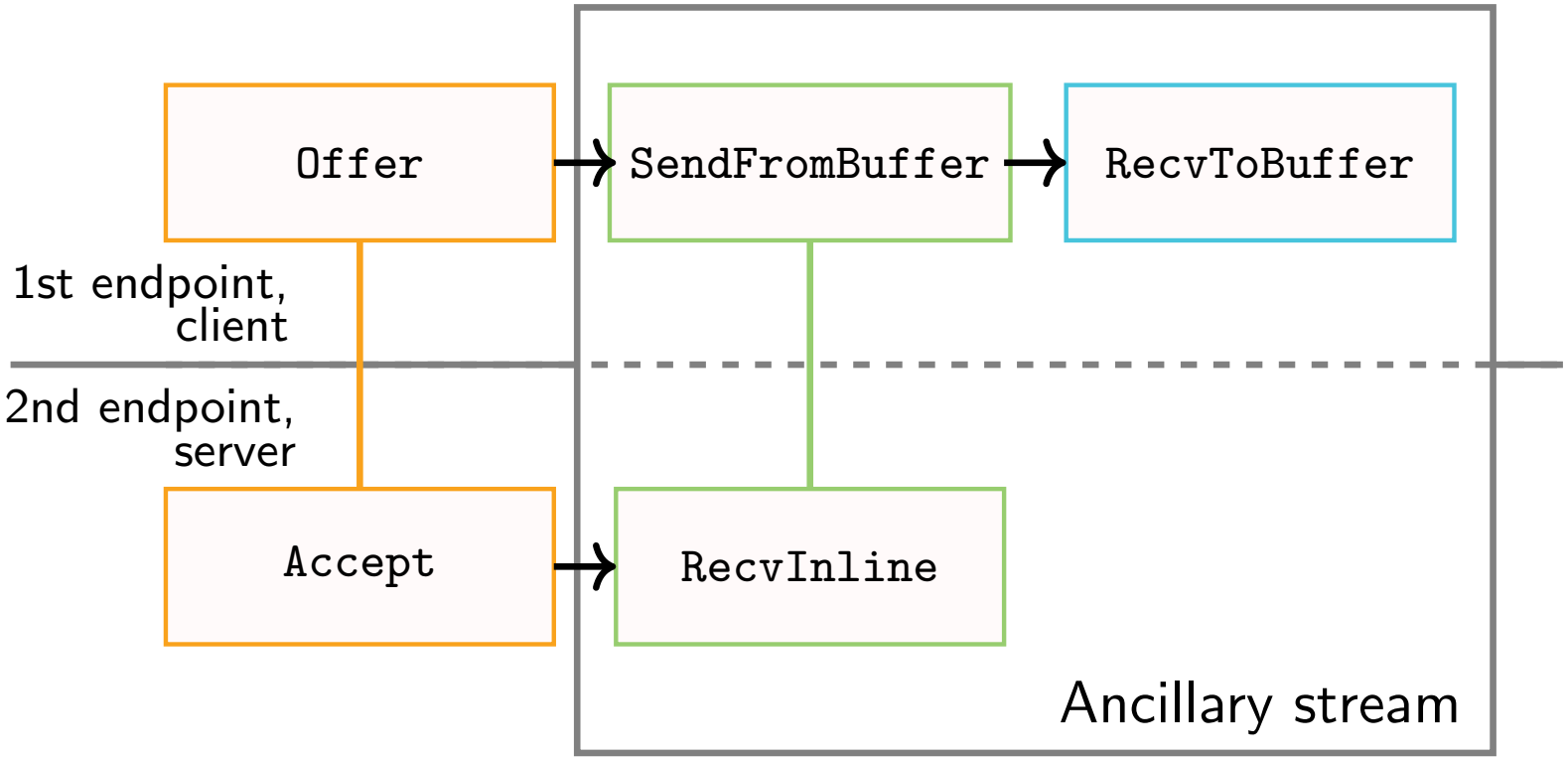
1. Server posts `Accept` → `RecvInline` (to receive a request).

Example: client/server IPC



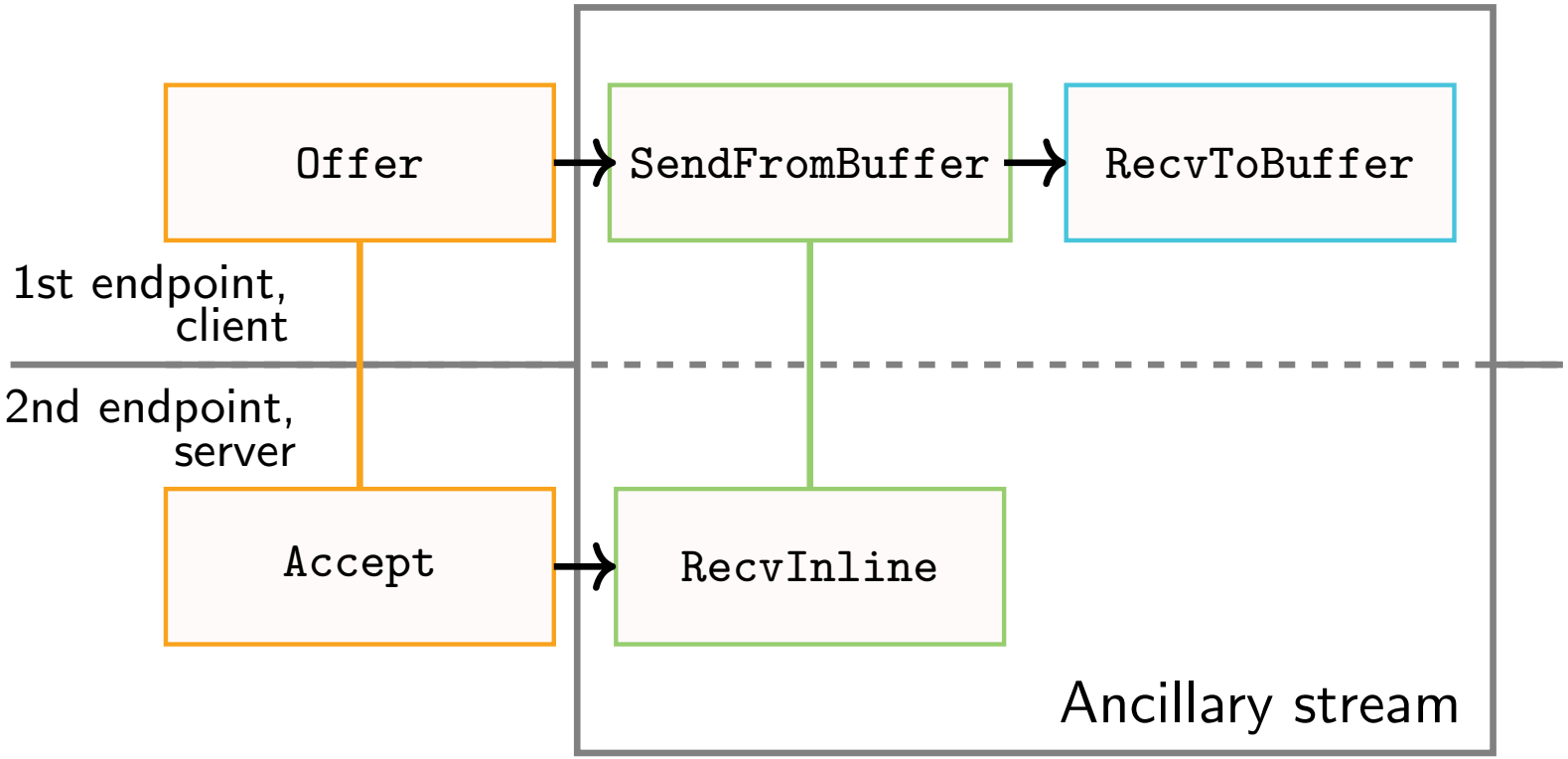
1. Server posts `Accept` → `RecvInline` (to receive a request).
2. Client posts `Offer` → `SendFromBuffer` → `RecvToBuffer` (to send a request and receive a response).

Example: client/server IPC



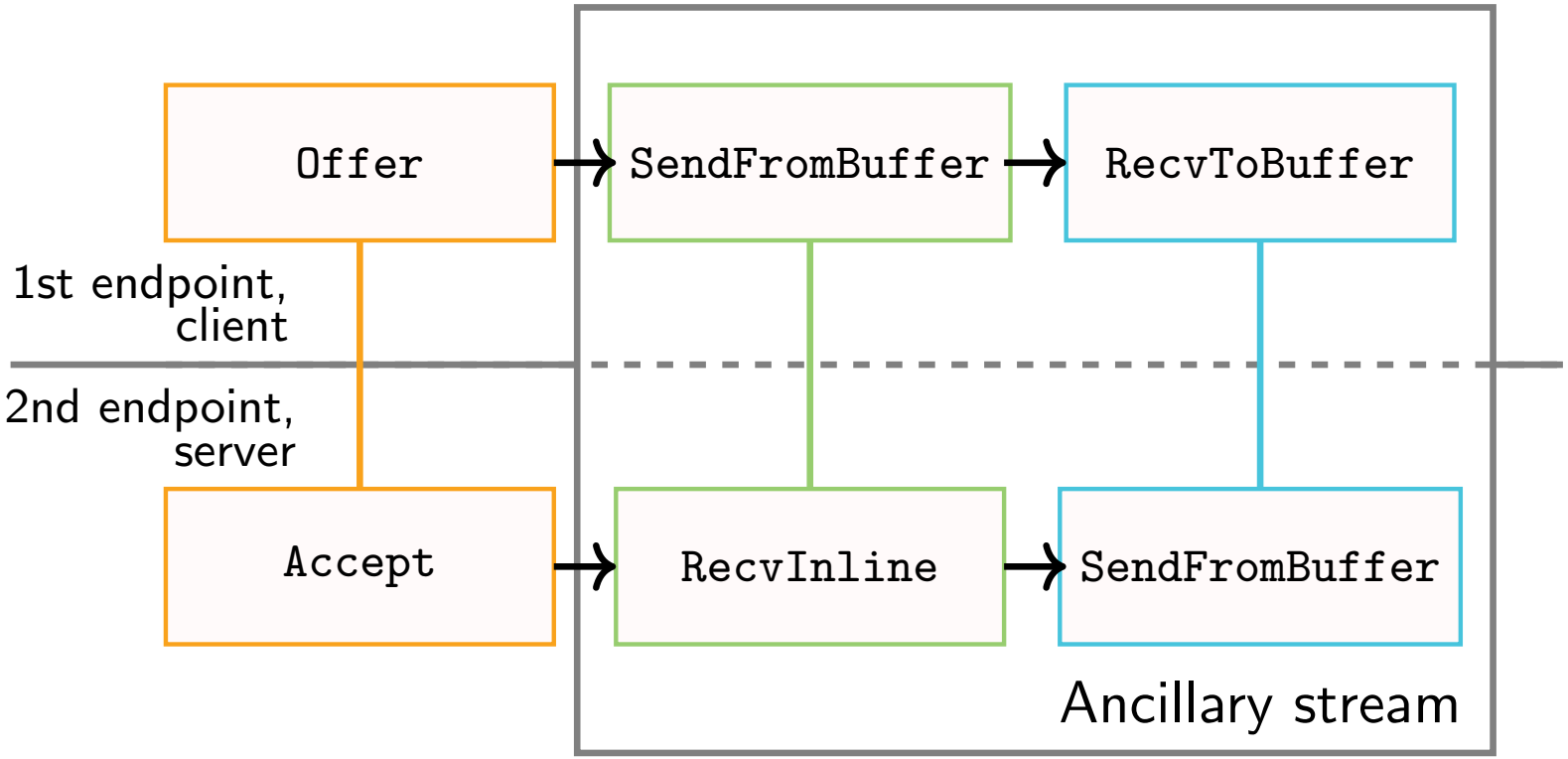
1. Server posts `Accept` → `RecvInline` (to receive a request).
2. Client posts `Offer` → `SendFromBuffer` → `RecvToBuffer` (to send a request and receive a response).

Example: client/server IPC



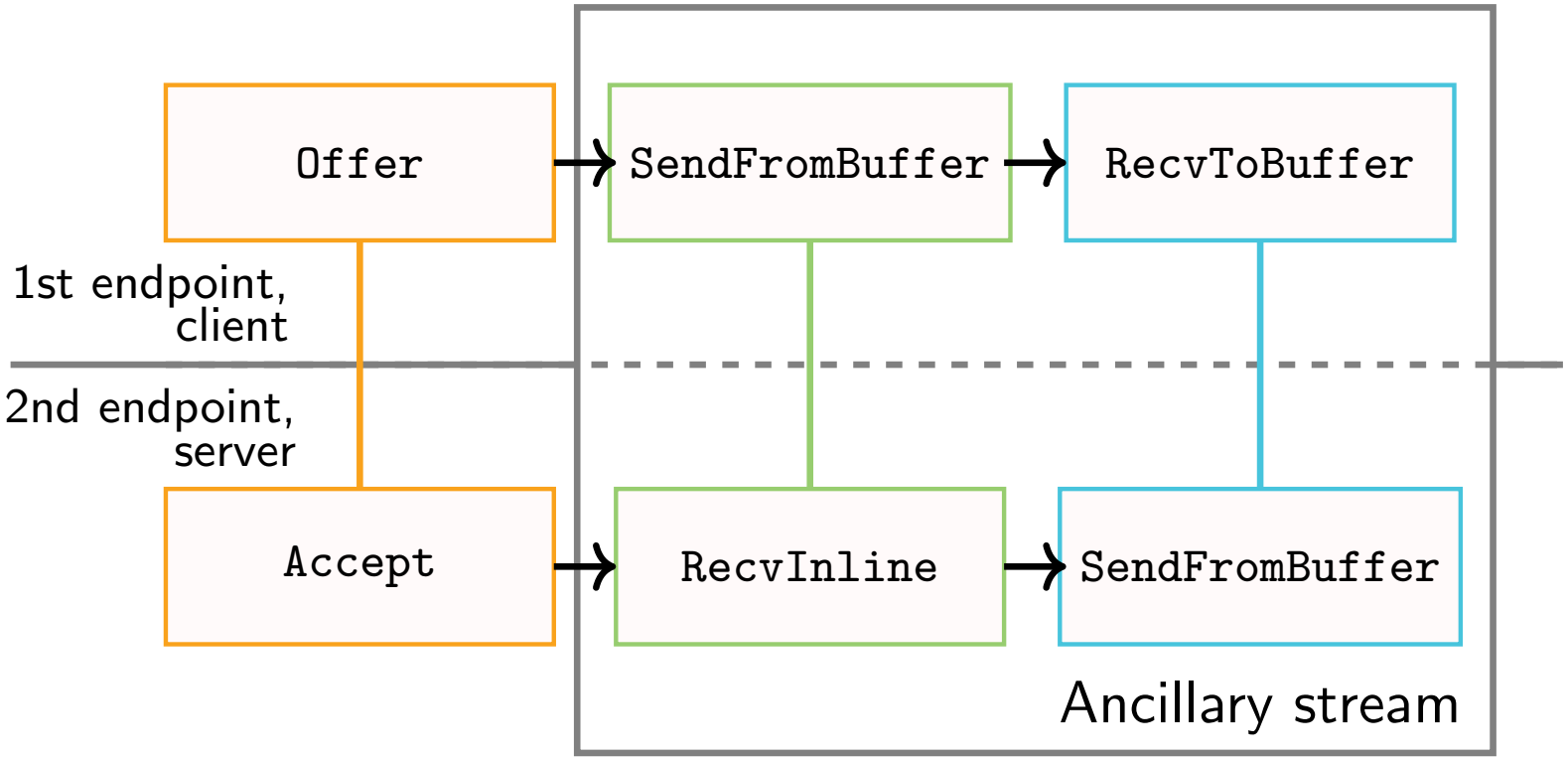
1. Server posts `Accept` → `RecvInline` (to receive a request).
2. Client posts `Offer` → `SendFromBuffer` → `RecvToBuffer` (to send a request and receive a response).
3. Server posts `SendFromBuffer` (to send the response).

Example: client/server IPC



1. Server posts `Accept` → `RecvInline` (to receive a request).
2. Client posts `Offer` → `SendFromBuffer` → `RecvToBuffer` (to send a request and receive a response).
3. Server posts `SendFromBuffer` (to send the response).

Example: client/server IPC



1. Server posts `Accept` → `RecvInline` (to receive a request).
2. Client posts `Offer` → `SendFromBuffer` → `RecvToBuffer` (to send a request and receive a response).
3. Server posts `SendFromBuffer` (to send the response).

Acknowledgements

Check out the paper for more details:

- Design of the kernel $\leftrightarrow$ user space ring buffer
- Memory management
- POSIX emulation

Acknowledgements

Check out the paper for more details:

- Design of the kernel $\leftrightarrow$ user space ring buffer
- Memory management
- POSIX emulation

Check out the project: `github.com/managarm/managarm`

Blog: `managarm.org` Twitter: `@managarm_OS`

Thanks to all contributors!

Acknowledgements

Check out the paper for more details:

- Design of the kernel $\leftrightarrow$ user space ring buffer
- Memory management
- POSIX emulation

Thank You! Questions?

Blog: `managarm.org` Twitter: `@managarm_OS`

Thanks to all contributors!