# Introduction

We know that different processes must run in separate spaces for security, stability, and memory management reasons:

* Security: Each process is sandboxed and run under a distinct system identity.
* Stability: If a process misbehaves (e.g. crashes), it does not affect any other processes.
* Memory management: Unneeded processes are removed to free resources (mainly memory) for new ones.

But in many cases, they want to communicate and share data with each other.

IPC is a technique for the exchange data across multiple processes.

There are many IPC mechanisms in Linux. Some of the most common are:

* Shared files
* Shared memory
* Pipes (named and unnamed)
* Message queues
* Sockets (UNIX socket)
* Signals
* RPC
* …

In this tutorial, we’ll discover all of these IPC mechanisms.

# System V vs POSIX

Both offer IPC mechanisms, but there are some differences:

|  |  |
| --- | --- |
| **System V** | **POSIX** |
| Older. Deprecated. | Newer |
| Covers all IPC mechanisms: socket, pipes (named and unnamed), message queues, signals, semaphores, and shared memory | Almost all basic concepts are the same as System V. It only differs with the interface. |
| Uses **keys and identifiers (integer numbers)** to identify IPC objects. | Uses **names (ASCII strings) and file descriptors** to identify IPC objects |
| When a process is killed forcefully, it **doesn’t release shared IPC resources (of shared memory, message queues, etc.) upon termination**.  The only exception is semaphore IPC which allows to be automatically released if process dies (using SEM\_UNDO flag). | Because of using reference counting technique, when a process is killed forcefully, the shared IPC resources will be **released correctly**. |
| N/A | **Multi-thread safe**. Covers thread synchronization functions such as mutex locks, conditional variables, read-write locks, etc. |
| N/A | Offers few notification features for message queues (such as mq\_notify()) |
| The size of shared memory segment is fixed at the time of creation (via shmget()) | The size of the underlying object can be adjusted with use ftruncate(), and then re-create the mapping using munmap() and mmap()) |

These APIs should never be mixed in a single application.

In a nutshell, **POSIX is better** and should be used in new projects.

**Note**: C++11 Standard Thread library does not include POSIX IPCs. It’s primarily focused on providing a high-level interface for managing threads and concurrent execution within a single process. If you’re using C++11 Standard Thread library and need some IPCs, you can freely choose System V or POSIX to be used with.

# Shared File

## What Is It?

Consider the simple case in which **one process (*producer*) creates and writes to a file**, and **another process (*consumer*) reads from this same file**.

A race condition might arise: the *producer* and the *consumer* might access the file at the same time, causing indeterminate behaviors. To avoid this, the **file must be locked** in a way that prevents a conflict between a write operation and any other operation, whether a read or a write.

The locking is managed by the OS. It can be summarized as follows:

* The *producer* should gain an ***Exclusive Lock*** on the file before **writing** to it. An Exclusive Lock can be held by one process at most, so no other process can access the file until the lock is released.
* The *consumer* should gain at least a ***Shared Lock*** on the file before **reading** it. Multiple readers can hold a shared lock at the same time, but no writer can access a file when a single reader holds a Shared Lock.

A Shared Lock promotes efficiency. If one process is just reading a file and not changing its contents, there is no reason to prevent other processes from doing the same. Writing, however, clearly demands Exclusive Lock to a file.

## Pros and Cons

**Pros:**

* Allow **lengthy, arbitrary bytes** (e.g., a digitized movie) because of using disk space.
* **Bidirectional**
* Only one shared file for multiple readers and multiple writers.
* Data stored in file is **permanent**. So even when the producer exited, the consumer can still read bytes from the file.

**Cons:**

* File access is **relatively slow**, whether the access involves reading or writing.
* Have to **synchronize manually** to prevent troubles.

## Example

*Directory: Linux\Code\IPCs\SharedFile*

The *producer* will write some data to a shared file, then the *consumer* will read the file and output to console. To protect the shared file, we’ll use the flock (in <fcntl.h> header) of POSIX.

The two programs should be executed in different terminals.

Here is the output from the two programs:

$ ./producer

Process 18822 has written to data file...

$ ./consumer

Hello World

# Shared Memory

## What Is It?

To synchronize access to the shared memory, a ***semaphore*** is used. In general, semaphore has two types:

* **Counting semaphore**: Has a value (typically initialized to zero) that can be incremented. Consider a bicycle-renting shop with a hundred of bicycles in stock. Every time a bike is rented, the semaphore is incremented by one; when a bike is returned, the semaphore is decremented by one. Rentals can continue until the value hits 100, but then must halt until at least one bike is returned, thereby decrementing the semaphore to 99.
* **Binary semaphore**: Is a special case requiring only two values: 0 and 1. In this situation, a semaphore acts as a **mutex**: a mutual exclusion construct. When the semaphore's value is 0, the writer alone can access the shared memory. After writing, this process sets semaphore's value to 1, thereby allowing the readers to read the shared memory.

## Pros and Cons

**Pros:**

* Memory access is **very fast**, whether the access involves reading or writing. Also, **no data transfer** happens in channel, making it even faster.
* **Bidirectional**
* Only one shared memory for multiple readers and multiple writers.

**Cons:**

* Have to **synchronize manually** to prevent troubles.
* Not good for massive data stream because of using **limited RAM**.
* Doesn't work across multiple machines.
* Data stored in memory is **volatile**. So when the writer exited, the reader cannot read bytes from the memory, because all data is removed from the memory.

## Example

### POSIX API

*Directory: Linux\Code\IPCs\SharedMemory\SingleSegment*

The *memwriter* writes some data to a shared memory segment, then the *memreder* reads the file and output to console. Once the *memwriter* exits, the *memreader* will fail to read data because the shared memory allocated by the *memwriter* was deleted.

The two programs should be executed in different terminals; first the *memwriter*, then the *memreader*.

Here is the output from the two programs:

$ ./memwriter

Shared mem address: 0x7fda029e8000 [0 ... 511]

Backing file: /dev/shm/shMemEx

$ ./memreader

Hello World

Note:

A downside of the POSIX API is that features are dependent upon the installed kernel version, which impacts code portability. For example, the POSIX API, by default, implements shared memory as a *memory-mapped file*: for a shared memory segment; the system maintains a backing file with corresponding contents. Shared memory under POSIX can be configured without a backing file, but this may impact portability. The backing file combines the benefits of memory access (speed) and file storage (persistence).

### System V API

<https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_shared_memory.htm>

<https://www.alexdelis.eu/k22/Rec3-ShrdMem-Sems.pdf>

# Pipe

## What Is It?

Pipes are channels which connect processes for communication. A channel has a *write end* for writing bytes, and a *read end* for reading these bytes in **FIFO** (first in, first out) order.

In typical use, **one process writes to the channel, and a different process reads from this same channel**.

The bytes themselves might represent anything: numbers, texts, digital movies, etc.

## Types of Pipes

### Unnamed Pipes

The vertical bar | (also called [pipe redirection](#_Pipe_Redirection_(|))) represents an unnamed pipe at the command line. For example:

$ sleep 5 | echo "Hello, world!" # writer to the left of |, reader to the right

The sleep and echo utilities execute as **separate processes**, and the unnamed pipe allows them to communicate.

The sleep is the writer, and the echo is the reader.

* The reader blocks until there are bytes to read from the channel.
* The writer – after writing its bytes (although in this example, the sleep does not write any bytes to the channel) – finishes up by sending an end-of-stream marker.

(Even if the writer terminates prematurely, an end-of-stream marker is sent to the reader.).

The unnamed pipe exits until both the writer and the reader terminate.

The **unnamed pipe has no backing file**: the system maintains an in-memory buffer to transfer bytes from the writer to the reader. Once the writer and reader terminate, the buffer is reclaimed, so the unnamed pipe goes away.

### Named Pipes

A named pipe has a backing file and a distinct API.

For example, here are the steps:

1. Open two terminals. The working directory should be the same for both.
2. In one of the terminals, run:

$ mkfifo tester # Create a backing file named "tester"

$ cat tester # Print the pipe's content to stdout

At the beginning, nothing should appear because nothing has been written yet to the named pipe.

1. In the second terminal, run:

$ cat > tester # Redirect keyboard input to the pipe

hello, world! # Then hit Return key

bye, bye # Then hit Return key

<Control-C> # Terminate session

Whatever is typed into this terminal is echoed in the other. Once Ctrl+C is entered, the regular command line prompt returns in both terminals: the pipe has been closed

1. Clean up by removing the file that implements the named pipe:

$ unlink tester

## Blocking vs Non-Blocking

Consider two processes, one reads data and another writes data. They're connected by a pipe, but speed of the data acquisition of two process is different (e.g., for each data, writing takes 10ms while reading takes only 5ms).

The **default behavior of pipes is blocking if the partner process (or partner end) is slower**. In this example, the reading process will have to wait for the writing process, making the whole process hang. This is **bad**!

If we want this not to happen, we have to change the pipe behavior to **non-blocking**. This way, the reading process won't wait (be blocked) for the writing process, but will keep running and might enter case of empty pipe (but this case won't harm the program if we handle it correctly).

## Pros and Cons

**Pros:**

* Pipes are very **fast**, although not as fast as shared memory. They're even faster than sockets when transferring small block sizes (e.g.: 100 bytes, 500 bytes, etc.)
* Automatically **synchronized**.
* FIFO behavior (the first byte written is the first byte read, the second byte written is the second byte read, and so forth) helps transfer data between processes very **stable** and **reliable**.
* Support **blocking** and **non-blocking**.
* Named pipes can provide communication between processes on **different machines** across a network, although they're not preferred as network sockets in this case.

**Cons:**

* Strict FIFO behavior makes pipe less flexible because **data cannot be retrieved out of FIFO order**.
* Pipe is byte-based stream, not message-based stream, making it **harder to work with messages**.
* **Unidirectional** (one-way communication**).**

## Examples

### Unnamed Pipe

*Directory: Code\IPC\Pipes\UnnamedPipe*

Here is the output from the program:

$ ./unnamed\_pipe

Hello World

### Named Pipe

*Directory: Code\IPC\Pipes\NamedPipe*

The two programs should be executed in different terminals with the same working directory. However, the *writer* should be started before the *reader*, as the former creates the pipe. The reader then accesses the already created named pipe.

Here is the output from the two programs:

$ ./writer

Opened pipe

Sent bytes: 1804289383 846930886 1681692777 1714636915

Sent bytes: 1957747793 424238335 719885386 1649760492

Sent bytes: 1189641421 1025202362 1350490027 783368690

Sent bytes: 1102520059 2044897763 1967513926 1365180540

Sent bytes: 304089172 1303455736 35005211 521595368

Sent bytes: 294702567 1726956429 336465782 861021530

Sent bytes: 233665123 2145174067 468703135 1101513929

Sent bytes: 1801979802 1315634022 635723058 1369133069

Sent bytes: 1059961393 2089018456 628175011 1656478042

Sent bytes: 1131176229 1653377373 859484421 1914544919

In total, 40 bytes sent to the pipe.

$ ./reader

Opened pipe

Received bytes: 1804289383

Received bytes: 846930886

Received bytes: 1681692777

Received bytes: 1714636915

Received bytes: 1957747793

Received bytes: 424238335

Received bytes: 719885386

Received bytes: 1649760492

Received bytes: 1189641421

Received bytes: 1025202362

Received bytes: 1350490027

Received bytes: 783368690

Received bytes: 1102520059

Received bytes: 2044897763

Received bytes: 1967513926

Received bytes: 1365180540

Received bytes: 304089172

Received bytes: 1303455736

Received bytes: 35005211

Received bytes: 521595368

Received bytes: 294702567

Received bytes: 1726956429

Received bytes: 336465782

Received bytes: 861021530

Received bytes: 233665123

Received bytes: 2145174067

Received bytes: 468703135

Received bytes: 1101513929

Received bytes: 1801979802

Received bytes: 1315634022

Received bytes: 635723058

Received bytes: 1369133069

Received bytes: 1059961393

Received bytes: 2089018456

Received bytes: 628175011

Received bytes: 1656478042

Received bytes: 1131176229

Received bytes: 1653377373

Received bytes: 859484421

Received bytes: 1914544919

In total, received 40 bytes.

### Non-Blocking Unnamed Pipe

Directory: *Directory: Code\IPC\Pipes\NonBlocking\_UnnamedPipe*

## Notes

* The processes, which handle a *blocking* pipe, shouldn't perform any other task except reading data from pipe and writing data to pipe.
* It's not necessary that a pipe must be created in the writer process. The reader process can also create the pipe. The key here is that **the process started first should be the one creating the pipe**.
* The communication channel provided by a pipe is a **byte stream**. There is **no concept of messages**.

If you want to handle messages with pipes, the writer must find a way to help the reader know how many bytes for each message (preventing missing content of messages). The idea is adding an item defining message length to the transferred message. Example: *Code\IPC\Pipes\PipeForMessage*

Another way, you should use a truly message-oriented approach, such as [Message Queue](#_Message_Queues) or [UNIX socket](#_IPC_Socket).

## Tips

* How big is the pipe buffer: <https://unix.stackexchange.com/a/11954>
* Change maximum size of pipe: <https://stackoverflow.com/a/4739461>
* Why use a named pipe instead of a file: <https://askubuntu.com/a/449192>

# Message Queues

## What Is It?

Pipes have strict FIFO behavior. By contrast, **message queues can behave in the same way but are flexible enough that byte chunks can be retrieved out of FIFO order**.

Details:

sender ---> |3| ---> |2| ---> |2| ---> |1| ---> receiver

If strict FIFO behavior were in play, then the messages would be received in the order 1-2-2-3. However, the message queue allows other retrieval orders; for example, 3-2-1-2.

As the name suggests, a message queue is a sequence of messages, each of which has two parts:

* A payload, which is an array of bytes (char in c)
* A type (given as a positive integer value) categorizes messages for flexible retrieval

## Blocking vs Non-Blocking

Similar to pipes, message queues can be either blocking or non-blocking.

## Pros and Cons

**Pros:**

* Message queues are very **fast**, although not as fast as shared memory.
* Automatically **synchronized**.
* Allow retrieving messages **out of FIFO order**.
* Support **blocking** and **non-blocking**.

**Cons:**

* **Unidirectional**

## Example

*Directory: Linux\Code\IPCs\MessageQueue*

The message queue example consists of two programs: the *sender* that writes to the message queue and the *receiver* that reads from this queue.

In this example, the sender sends the messages in the order 1-1-2-2-3-3, but the receiver then retrieves them in the order 3-1-2-1-3-2.

Here is the output from the two programs:

$ ./receiver

msg5 received as type 3

msg1 received as type 1

msg3 received as type 2

msg2 received as type 1

msg6 received as type 3

msg4 received as type 2

$ ./sender

msg1 sent as type 1

msg2 sent as type 1

msg3 sent as type 2

msg4 sent as type 2

msg5 sent as type 3

msg6 sent as type 3

## Tips

<https://www.geeksforgeeks.org/sharing-queue-among-three-threads/>

<https://www.geeksforgeeks.org/condition-wait-signal-multi-threading/>

# UNIX Socket

## What Is It?

*UNIX sockets* (aka Unix Domain Sockets) enable channel-based **communication for processes on the same physical device** (host). They rely upon the local system kernel to support communication; in particular, communicate using a local file as a socket address.

The server and client can run on the same machine. In this case, the server uses network address *localhost* (e.g., 127.0.0.1).

Further info:

*Network Sockets* are another type of IPC socket which are used for processes that can run on different physical devices. They need support from network interfaces and underlying IP protocols such as **TCP** (Transmission Control Protocol) or **UDP** (User Datagram Protocol).

By contrast, UNIX socket uses local files to send and receive data instead of network interfaces and IP packets.

Despite these implementation differences, UNIX socket and network socket APIs are the same in the essentials.

## Blocking vs Non-Blocking

Similar to pipes, sockets can be either blocking or non-blocking.

## Socket Details

For more insights to many different concepts of socket (such as domain type, underlying protocol, port binding, Linux APIs, etc.), check one of my tutorials: *Tutorials\Linux\Sockets.docx*

## Pros and Cons

**Pros:**

* Faster than pipes when transferring large block sizes (e.g.: 10 Kbytes, 1 Mbyte, etc.)
* Automatically **synchronized**.
* Support **blocking** and **non-blocking**.
* Easy to convert a UNIX socket to a network socket to work **across multiple machines**.
* Easy to expose process as a **service**.
* Transfer data between processes **very stable** and **reliable.** With TCP, the server ensures that bytes are transferred in the correct order; and if it fails, it will continue retrying until success.

**Cons:**

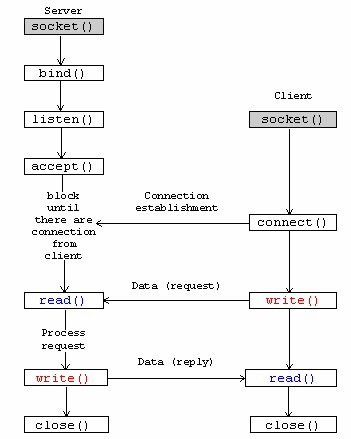
* UNIX sockets are **one-to-one**. You need multiple sockets if you want to send the same thing to multiple processes.

## Example

*Directory: Linux\Code\IPCs\UnixSocket\TCP*

The socket example consists of two programs: a *server* and a *client*. The server creates a localhost TCP socket, then waits for requests from the client. Each time, the server receives a request from the client, it responses "OK" to the client.

The flow is basically as follows:



Here is the output from the two programs:

**$ ./server**

Server binded its socket to address 2130706433 and port 60020 # 2130706433 = "127.0.0.1"

Server is listening clients...

Server accepted client connection

Client request: War and Peace

Server response: OK

Client request: Pride and Prejudice

Server response: OK

Client request: The Sound and the Fury

Server response: OK

**$ ./client**

Client connected to server at IP 2130706433, about to write some stuff...

Client request: War and Peace

Server response: OK

Client request: Pride and Prejudice

Server response: OK

Client request: The Sound and the Fury

Server response: OK

# Signal

## What Is It?

A signal interrupts an executing program and, in this sense, communicates with it. Most signals can be either ignored (blocked) or handled, with SIGSTOP (pause) and SIGKILL (terminate immediately) as the two notable exceptions.

Signals can arise in user interaction. For example, a user hits Ctrl + C, which generates a SIGTERM signal, from the command line to terminate a program started from the command-line. Unlike SIGKILL, SIGTERM can be either blocked or handled.

One process also can signal another, making signals an IPC mechanism.

## Pros and Cons

**Pros:**

* Straightforward

**Cons:**

* Very **limited number of use cases** compared to other IPC mechanisms.

## Example

*Directory: Linux\Code\IPCs\Signal*

The *shutdown* program simulates the graceful shutdown of a multi-processing system, in this case, a simple program consisting of a parent process and a child process. The parent process kills the child gracefully.

**$ ./shutdown**

Parent is sleeping so that the child can execute for a while...

Child just woke up, but going back to sleep.

Child just woke up, but going back to sleep.

Child just woke up, but going back to sleep.

Child just woke up, but going back to sleep.

Child confirming received signal: 15

Child about to terminate gracefully...

Child terminating now...

My child terminated, about to exit myself...

# RPC

## What Is It?

RPC (Remote Procedure Call) is a technique for constructing distributed client-server based applications.

It’s developed by Sun Microsystems. It extends the conventional local procedure calling so that **the called procedure need NOT exist in the same address space as the calling procedure**. The two processes **may be on the same system**, or they **may be on different systems** with a network connecting them.

|  |  |
| --- | --- |
| Lightbox | The simple model is:  1. The procedure parameters are transferred across the network to the environment where the procedure is to execute.  2. After the procedure finishes executing, the results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call. |

## How It Works?

First, we should understand following terms:

* **Client**: The program on a system requesting the execution of code.
* **Server**: The program accepting the request from the client where the remote code is to be executed.
* **Stub**: Portion of code to convert addresses and parameters between client and server systems. The stub can handle conversion of param types between systems which have a different OS. The functions used by the stub can convert params from integers to floating point decimals and vice versa.
* **Marshalling (sắp xếp theo thứ tự)**: Placing parameters and addresses in a message to send to the server.
* **Unmarshalling**: Removes parameters and addresses from a message when received by the server to be sent to the RPC Server Service.
* **RPC Server Service**: The component in the server handles RPC requests from the client and executing the corresponding procedures or functions.

Now, following steps take place during a RPC :

|  |  |
| --- | --- |
| Lightbox | 1. A client invokes a client stub procedure, passing params as usual. The client stub resides within the client’s own address space. 2. The client stub marshalls (pack) the params into a message. Marshalling includes converting the representation of the params into a standard format, and copying each param into the message. 3. The client stub passes the message to the transport layer, which sends it to the remote server machine. 4. On the server, the transport layer passes the message to a server stub, which demarshalls (unpack) the params and calls the server routine using the regular procedure call mechanism. 5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer. 6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub. 7. The client stub demarshalls the return params and execution returns to the caller. |

**Network Protocol:**

RPC packagets are usually sent over TCP/IP, but can be sent over other network protocols. TCP/IP is used over the Internet, while some LANs or WANs may use another protocol.

**Binding:**

*How does the client know who to call, and where the service resides?*

The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

## Pros and Cons

**Pros:**

* Provides abstraction (message-passing nature of network communication is hidden from the user).
* Often omits many of the protocol layers to **improve performance**. Even a small performance improvement is important because a program may invoke RPCs usually.
* Enables the usage of the applications in the **distributed environment**, not only in the local environment.
* Minimizes re-writing / re-developing effort.
* Supports process-oriented and thread oriented models.

**Cons:**

* Not included in standard Linux library.

## Example

While other IPCs (pipes, sockets, shared memory, message queues, etc.) are low-level, RPC is a high-level IPC which is typically implemented as a separate layer on top of other IPCs.

So have to use third-party library, such as [gRPC](https://github.com/grpc/grpc), SunRPC (ONC RPC), etc.

## Tips

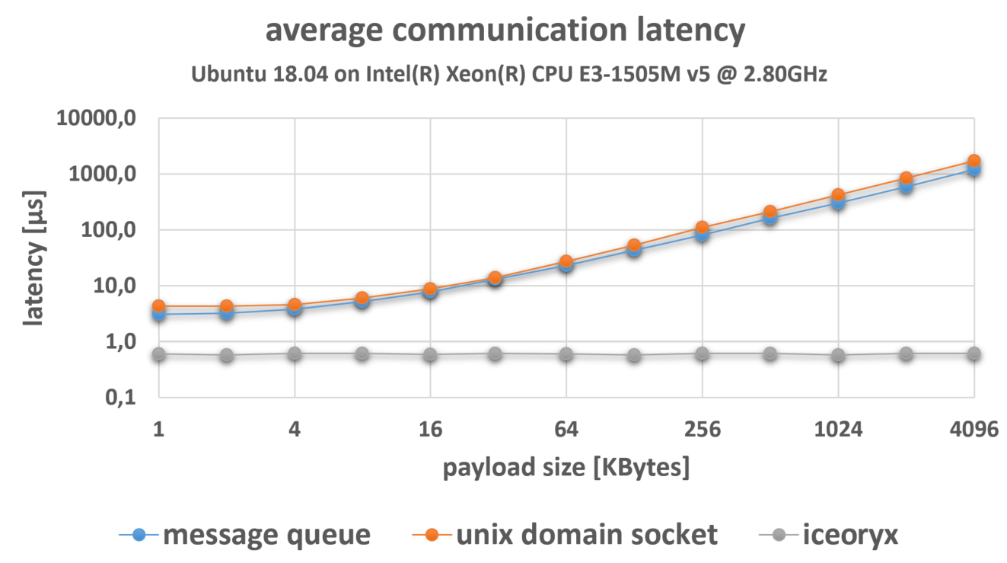
* To know what RPC services your Linux computer has, run: $ rpcinfo -p localhost.

# Iceoryx

## What Is Iceoryx?

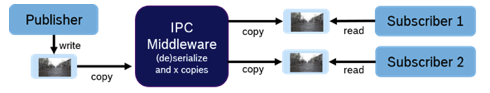
Eclipse Iceoryx is an IPC middleware for various OSs (currently Linux, macOS, QNX, FreeBSD and Windows 10), designed and implemented by Robert Bosch Corp.

It uses **True Zero-Copy** – a **shared-memory** approach that allows to transfer data from publishers to subscribers without a single copy. This ensures data transmissions with constant latency, regardless of the size of the payload.



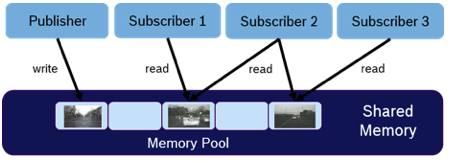
## How It Works?

A copy perspective of a typical IPC middleware solution:



When speeds reach GB/s, every message copy has a significant cost in terms of runtime and latency.

By contrast, true zero-copy communications:



Iceoryx is based on shared memory, which is nothing new. But it takes the approach further, combining with a **publish/subscribe architecture**, **service discovery**, **modern C++**, and **lock-free algorithms**.

A publisher writes the message directly to a chunk of memory that was previously requested from the middleware. When the message is delivered, subscribers receive references to these memory chunks while maintaining their own queue with a configurable capacity.

Every subscriber can have a unique view of which messages are still in process and which can be discarded. **References** are counted and memory chunks are released when they have no readers left .

One important aspect is that publishers can write again while subscribers are still reading because there is no interference from subscribers. The publisher is simply allocated a new memory chunk if the previous one is still in use.

If a subscriber is operating in polling mode and chunks are queued up until the subscriber checks the queue again, we can recycle older memory chunks using the **lock-free queue** in a process we call "safely overflowing."

The lock-free queue allows us to guarantee a memory-efficient contract is made with the subscriber with respect to the maximum number of latest messages that are stored in the queue, no matter how long the time between successive subscriber polls. This is a very helpful approach in common use cases such as those with a high-frequency publisher and a subscriber that is only interested in the latest, greatest message.

Because it is simply passing around **smart pointers**, iceoryx enables data transfers without actually transferring the data.

However, because the **message payload is not serialized**, a message must have the same memory layout for publishers and subscribers. For IPC on a specific processor, this can be ensured by using the same compiler with the same settings.

The message cannot contain any pointers to memory within the process’ internal virtual address space. This restriction also applies to heap-based data structures. If these constraints cannot be fulfilled, iceoryx can still be used with a top-level layer that handles the serialization in and the deserialization from the shared memory. In this case, iceoryx handles the low-layer transport that creates no copy itself.

Iceoryx depends on the **POSIX API**. Currently it supports Linux and QNX as underlying OSs. Because there are sometimes slight API differences, small adaptions might be necessary when porting iceoryx to another POSIX-based operating system.

Full guideline: <https://iceoryx.io/latest/>

## Pros and Cons

**Pros:**

* Message transfers with a latency of less than 1 µs
* Made to handle GBytes/sec data transfers

**Cons:**

* Very high-level, so difficult to customize

## Example

<https://iceoryx.io/latest/examples/icedelivery/>

# Tips

## Show IPC Info

The ipcs command shows information on the inter-process communication facilities

<https://www.geeksforgeeks.org/ipcs-command-linux-examples/>

**Tips:**

1. Clean up shared memory, semaphore, message queue:

$ ipcs -m | grep `whoami` | awk '{ print $2 }' | xargs -n1 ipcrm -m # shared memory

$ ipcs -s | grep `whoami` | awk '{ print $2 }' | xargs -n1 ipcrm -s # semaphore

$ ipcs -q | grep `whoami` | awk '{ print $2 }' | xargs -n1 ipcrm -q # message queue