EECS 4314 - Bit Theory
Concrete Architecture Report

Amir Mohamad
amohamad@my.yorku.ca

Arian Mohamad Hosaini
mohama23@my.yorku.ca

Dante Laviolette
dantelav@my.yorku.ca

Diego Santosuosso Salerno
nicodemo@my.yorku.ca

Isaiah Linares
isaiah88@my.yorku.ca

Joel Fagen
joefagan@my.yorku.ca

Misato Shimizu
misato1@my.yorku.ca

Muhammad Hassan
furquanh@my.yorku.ca

Yi Qin
aidenqin@my.yorku.ca

Zhilong Lin
lzl1114@my.yorku.ca

York University

March 10, 2023

Abstract

Concrete Architecture identifies the relationships between the subsystem and components inside the project. It includes what the system does, how it will implement the functionality, identification of significant components and connectors and realization of architecture mechanisms. The concrete architecture will investigate the responsibilities, component interaction, control and data flow. Interprocess communication is the methodology that allows the processes to communicate with each other and synchronize their actions. This communication between the processes can also be called a method of cooperation between processes. There are several advantages of using IPC. First, it is convenient because it allows a single user to work on many tasks simultaneously. IPC can divide the extensive system's functions into small processes or threads. These small processes can also speed up the processes of each execution and simultaneously execute the other processes. These processes will share the data they are using concurrently, reducing the time for processes to reload the data into the memory. In FreeBSD, the IPC model will provide access to communication networks such as the Internet. For 4.2BSD, the interprocess-communication facilities were intended to provide a sufficiently general interface to allow network-based applications to be constructed independently of the underlying communication facilities. To allow multiprocess programs, FreeBSD uses UNIX pipe. These pipes force systems to be designed with a contorted structure. The pipe contains four properties to interact with the data being used in the system. The four properties are:

- In-order delivery of data.
- Unduplicated delivery of data.
- Unduplicated delivery of data.
- Reliable delivery of data and connection-oriented communication.

In FreeBSD, the system must support communication networks that use different sets of protocols, naming conventions, hardware, and so on. In this case, the IPC contains the communication domain. A communication domain embodies the standard semantics of communication and naming, allowing the system to connect to different networks with different standards. To communicate across networks, FreeBSD uses sockets. Compared to the pipe method, sockets have not only the properties pipes have, but they also have properties such as preserving message boundaries and supporting out-of-band messages.

Keywords— concrete architecture, conceptual architecture, IPC, FreeBSD, pipes, sockets

Contents

1 Introduction

As we have discussed, the operating system is one of the essential parts of all kinds of computers, and it is responsible for managing the whole computer system. Due to this complex mission, the operating system needs the support of several subsystems to help it finish the job perfectly. One of those main subsystems is the interprocess communication system, also known as IPC, which is the target we will discuss. Interprocess communication can be defined as the mechanism that the operating systems provide to manage the data shared between processes in the operating system. The principle of interprocess communication is pretty simple. While the processes or applications are trying to transfer data, the process that requests the data will be considered a client, and the process that sends the data will be the server. Most of the applications can be both client and server. There are three goals for the interprocess-communication model. The first was providing access to communication networks such as the Internet. The second goal was to allow multiprocess programs, such as distributed databases, to be implemented. The third goal is to provide new communication facilities for constructing local-area network services, such as file servers. These facilities are designed to support: Transparency which ensures that the communication between processes can be on different machines, Efficiency, which provides the applicability of any IPC facility and Compatibility ensures the existing naïve processes should be usable in a distributed environment without change.

1.1 Overview

The report will analyze the concrete architecture of the FreeBSD operating system's interprocess communication(IPC) system. We will deeply explore the derivation process and learn about the architecture of IPC. After that, the report will analyze the reflection and use case to help understand IPC from another point of view. Finally, there will be a conclusion summarizing the limitations we met in the exploration and the lesson we have learnt from this process.

2 Derivation Process

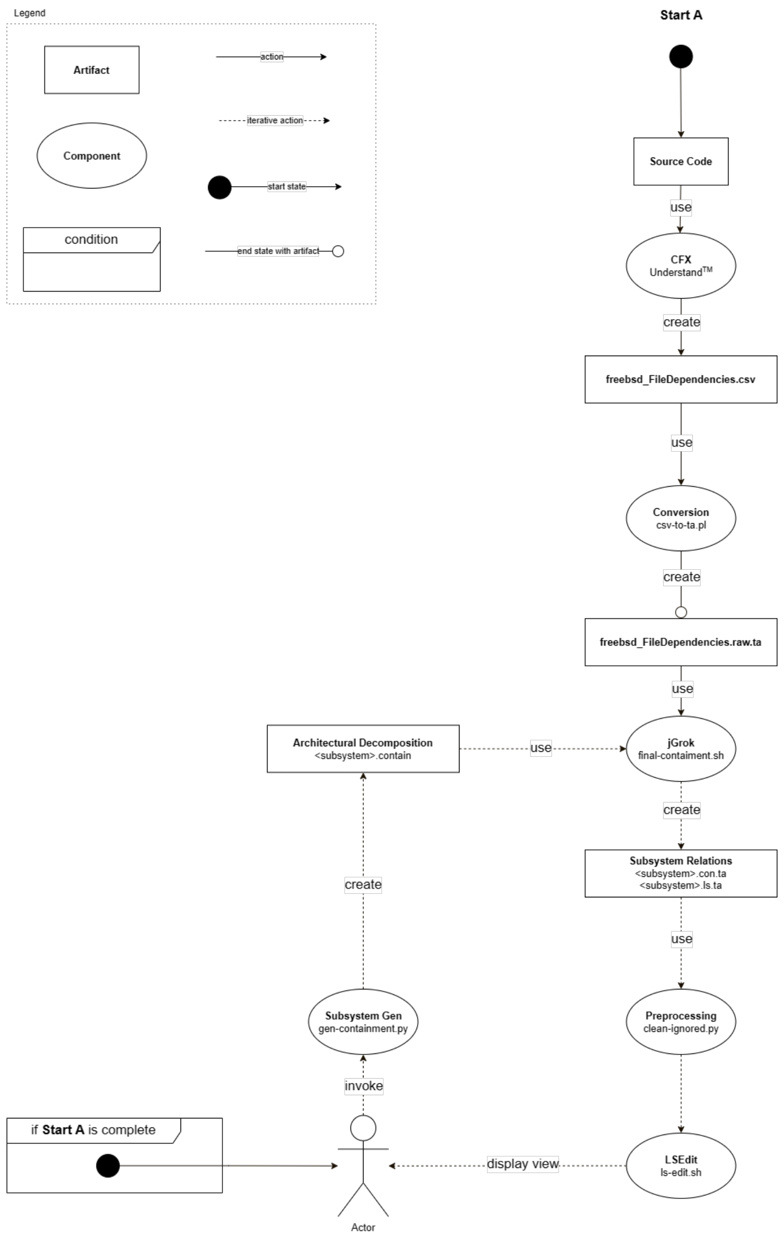


Figure 1: Concrete architecture derivation process [2]

Unlike the process of understanding conceptual architecture, which requires mainly documentation reviews, understanding concrete architecture can be more challenging and complicated as it requires source code observations. To have better understanding of FreeBSD's concrete architecture, we followed extraction steps, and they are shown as a derivation process in a figure above. After downloading FreeBSD source code, we use a source code extractor (`cfx`), Understand, which retrieves relations from the source code (e.g., control and data flow dependencies, relations among function calls). Here, we obtain a file, `freebsd_FileDependencies.csv` which has been produced by Understand. By running a Perl command to convert the csv file into a `raw.ta` file using `csv-to-ta.pl`, software, called `jGrok`, figures out their subsystem relations. After filtering dependencies by running `clean-ignored`.py, a graphical landscape editor, `LSEdit`, takes a role of visualizations for the extracted relations. Our task here is to manually observe those visualized diagrams using `ls-edit.sh`, and decompose subsystems hierarchically and pass the decomposed architectural structure to `jGrok`. We repeat these processes until we reach the point where the subsystem structure is understood in clear and detailed manners.

3 Results

3.1 Conceptual Architecture

Figure 2 below is the conceptual architecture derived in assignment 1. We won't go into much detail but we will have it as reference as we analyze the concrete architecture in comparison.

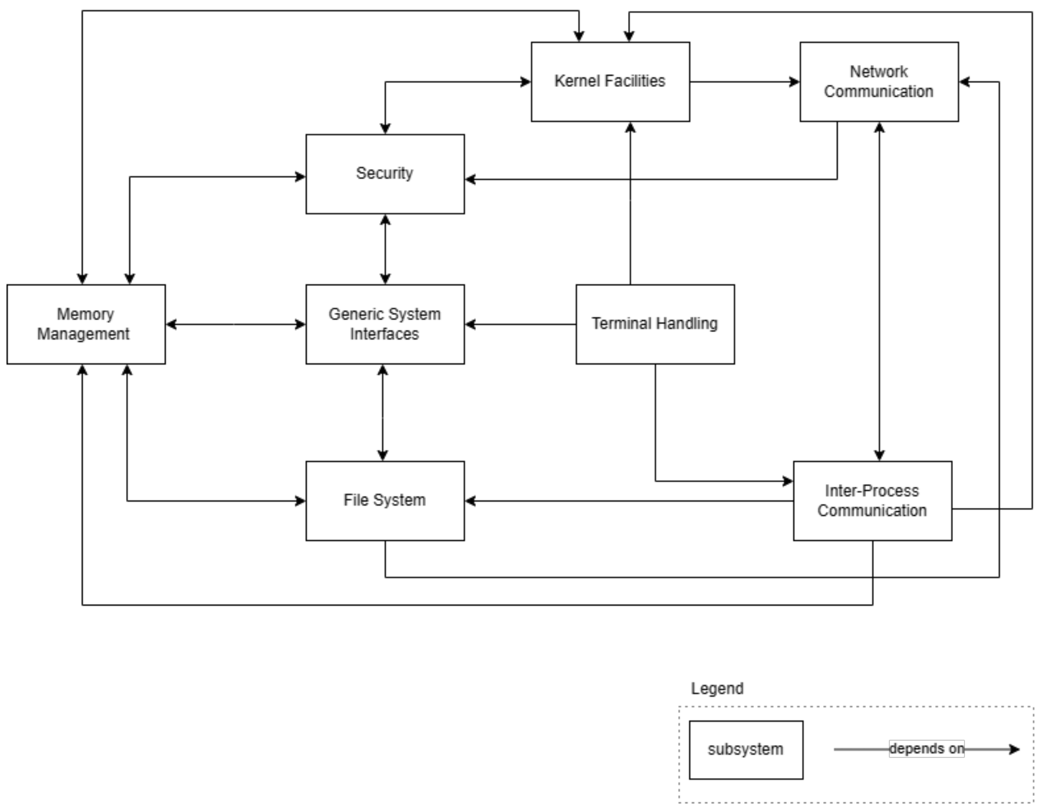


Figure 2: Conceptual architecture of FreeBSD subsystems

3.2 Concrete Architecture

The final iteration of our derivation process resulted in a concrete dependency diagram of the FreeBSD subsystems in Figure 3. Notably, we discovered several new dependencies represented by dashed red lines and unexpected bidirectional dependencies, depicted by hollow arrows. We also discovered a few removed dependencies, represented by crossed arrows. Furthermore, we identified two new subsystems, Crypto and Subroutines. To our surprise, majority of our existing dependencies in our conceptual diagram in Figure 2 stayed consistent, besides the various new divergences. Later in the report, we will analyze a few of these divergences in the IPC subsystem.

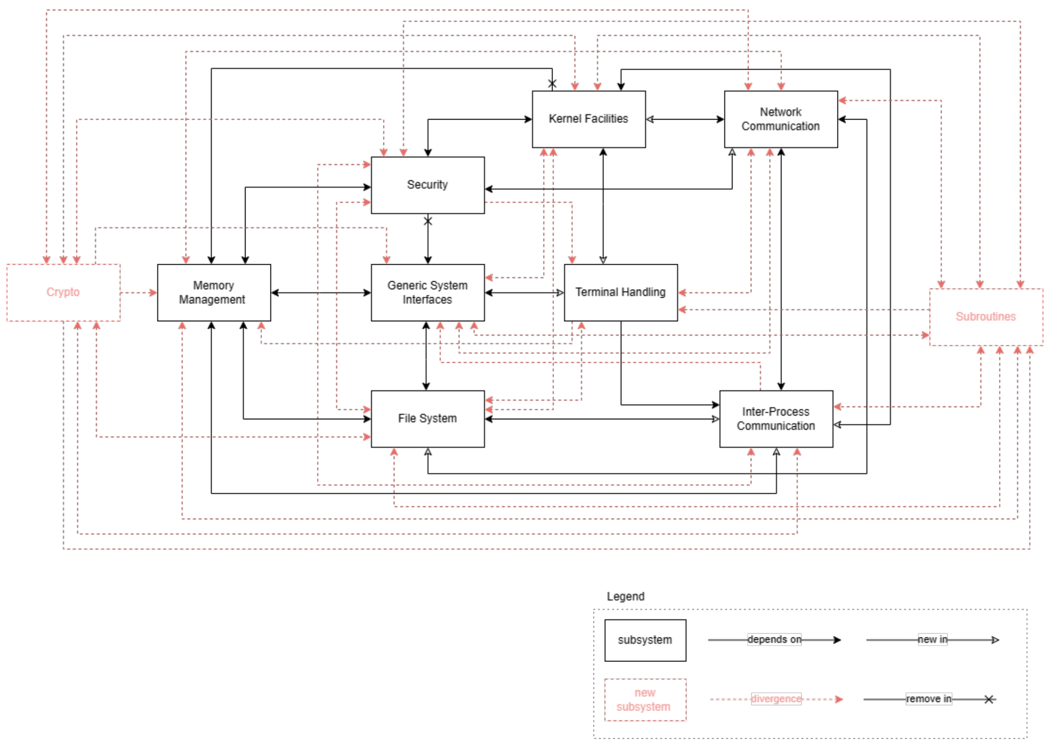


Figure 3: Concrete architecture of FreeBSD subsystems

3.3 Concrete Inter-Process Communication Architecture

The derivation process of the IPC subsystem in FreeBSD revealed a few new dependencies and subsystems. We will review some of the subsystems that IPC depends on and their functionality. The Security subsystem supports IPC, performing security audits on IPC mechanisms to ensure the system is traceable and secure [3]. The Generic System Interfaces subsystem provides the necessary data structures for implementing IPC. For example, IPC uses the message queue data structure defined in `sys/msg.h` and the shared memory data structure `sys/shm.h` to represent data for various implementations logically. The new Subroutines subsystem offers support functions for complex tasks. For instance, IPC uses system calls indirectly using `subr_syscall.c` when interacting with the kernel. Additionally, IPC depends on Crypto, which indirectly utilizes its cryptography functions to provide secure communication by encrypting and decrypting messages between processes. Understanding these dependencies provides valuable insights into the workings of the complex monolithic architecture of the FreeBSD kernel.

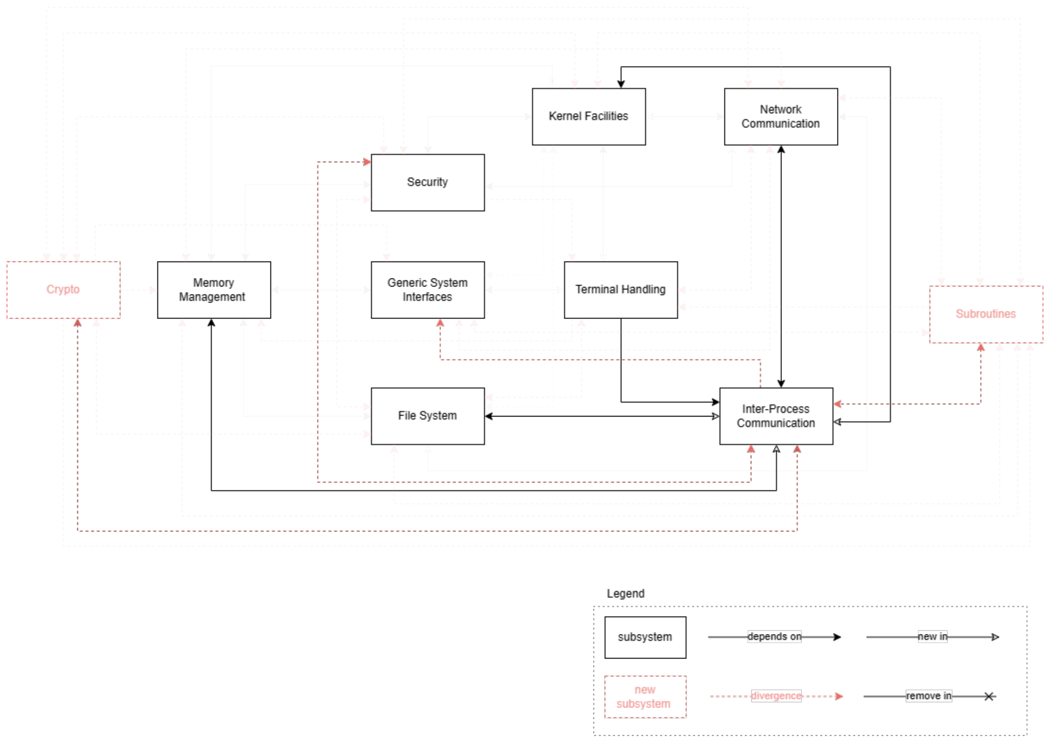


Figure 4: Concrete architecture of FreeBSD IPC subsystem

3.4 Top Level Kernel Dependencies

We created a custom script, `count-links.py` as a part of our derivation process to see how the FreeBSD kernel depended on the top level modules. We found that the kernel depended on the top level modules `lib`, `stand`, and `contrib` very strongly. All these dependencies had 1k to 6K+ use cases in the kernel. The kernel depended on the `stand` module, where the boot loader sources reside, the most with over 10k links.

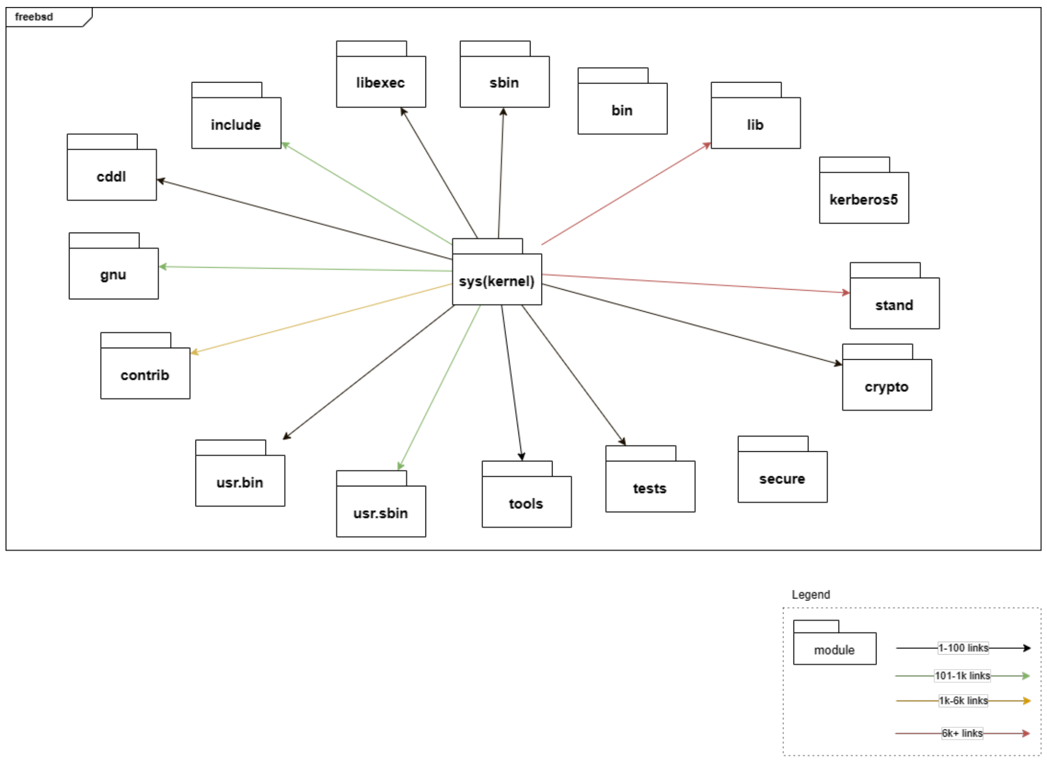


Figure 5: Kernel dependencies on top level modules in FreeBSD

4 Architecture

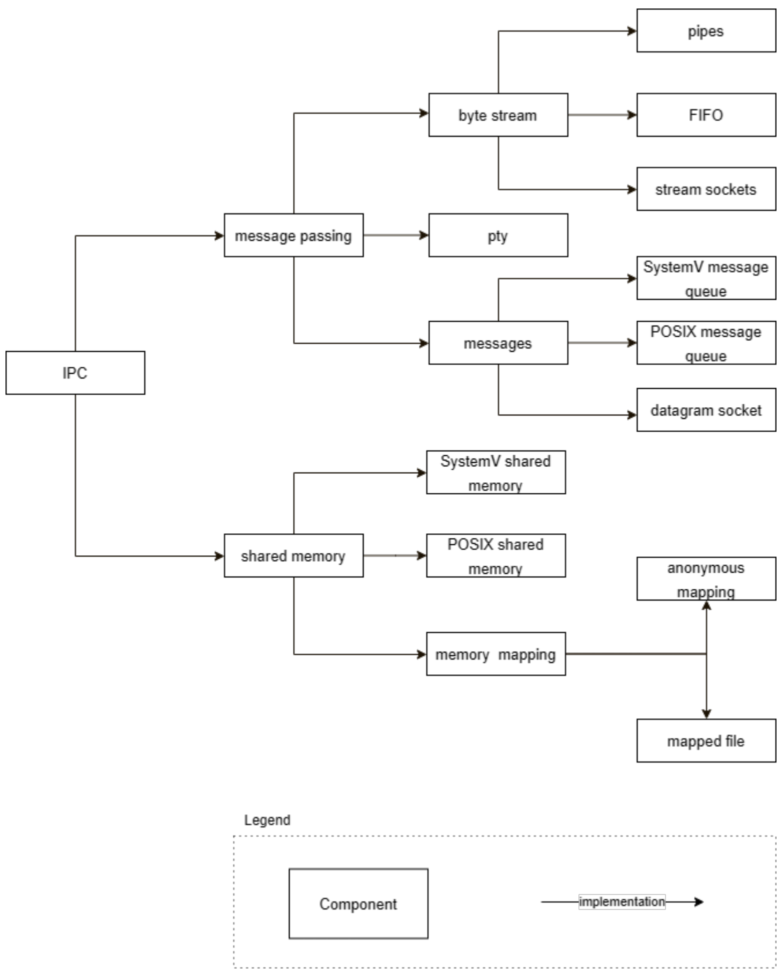


Figure 6: The IPC implementations breakdown of FreeBSD

4.1 Networking Inter-Process Communication

The network IPC facilities are layed on top of the networking facilities, and this heavy coupling is shown in the conceptual architecture and confirmed in the concrete architecture we derived. Data from the application is sent through the socket layer to the networking layer, and data coming in flows from the networking layer through the socket, and then into the receiving application. State required by the socket layer is fully encapsulated within it, whereas any protocol-related state is maintained in data structures that are specific to the supporting protocols. The system-call interface routines manage the actions related to a system call, collecting parameters, and converting user data into the format expected by the socket layer routine. Socket layer routines directly manipulate socket data structures and manage the synchronization between asynchronous activities.

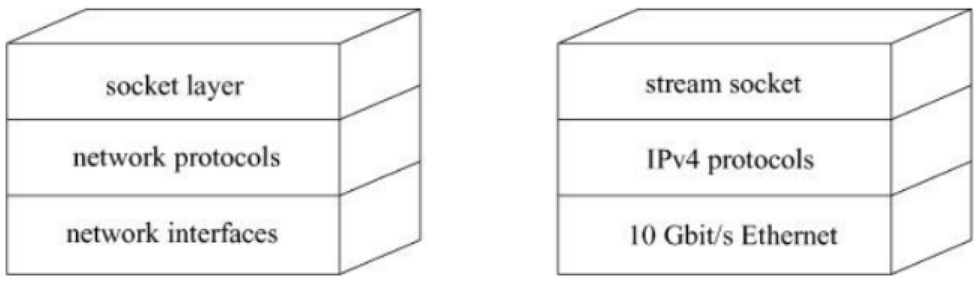
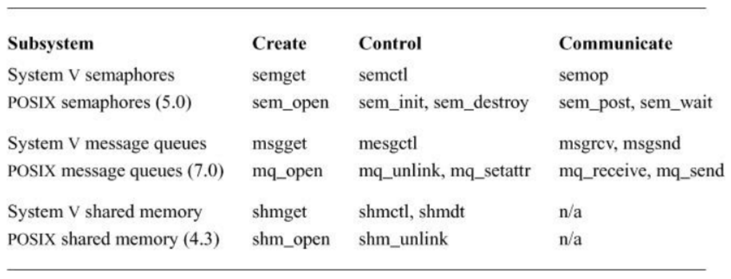


Figure 7: Networking IPC Layered Architecture
[12]

4.2 Local Inter-Process Communication

As well as network sockets, local interprocess communication is provided inside the kernel, in the form of semaphores, shared memory, and message queues. Initially, these mechanisms were provided by `System V`, and they are often referred to as `System V` semaphores, mutexes, and shared memory. There are some applications that use the `System V` implementation, such as `PostgreSQL` or `X11`. Later, `POSIX` was added to the kernel, and improves upon System V by introducing file descriptors. `POSIX`'s implementations of semaphores and shared-memory objects have replaced `System V`'s implementations in many applications. For example, Capsicum uses its own version of the `POSIX` shared-memory interface which links a file descriptor with an anonymous `vm_object` and allows multiple processes to share this object through the file descriptor.

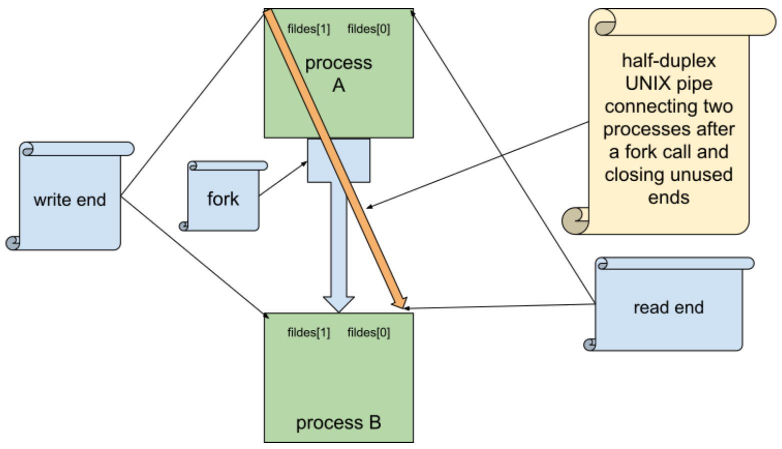
Previously, UNIX used the filesystem namespace to allow processes to rendezvous with each other and share their resources. But FreeBSD used the `System V` IPC, which introduced a new key-based namespace. Using `ftok()`, it would generate a key for a process based on its pathname. Multiple processes with the same pathname would get the same key. Using that key, a process would then create or get a specific object using the get call, using the `IPC_CREAT` flag to specify whether the object is being created or retrieved. The table below shows the `POSIX` and `System V` APIs for how they handle IPC objects.



Subsystem	Create	Control	Communicate
System V semaphores	semget	semctl	semop
POSIX semaphores (5.0)	sem_open	sem_init, sem_destroy	sem_post, sem_wait
System V message queues	msgget	mesgctl	msgrcv, msgsnd
POSIX message queues (7.0)	mq_open	mq_unlink, mq_setattr	mq_receive, mq_send
System V shared memory	shmget	shmctl, shmdt	n/a
POSIX shared memory (4.3)	shm_open	shm_unlink	n/a

Figure 8: Comparison of APIs between `System V` and `POSIX`
[12]

Pipes come from the `POSIX` standard, and are a common way to set up an interprocess communication channel for one process to send data to another in a byte-stream. Pipes are unidirectional, meaning they can only send data from one process to another and not in reverse. Contrast this with sockets (also in the `POSIX` standard), where data flow is bidirectional. A group of processes, aka a job, are connected by the shell via pipes. The output of the first process is fed into the second process, then the output of the second process is fed into the third, and so on. This is often referred to as a pipeline, and is reminiscent of the pipe and filter architecture style.



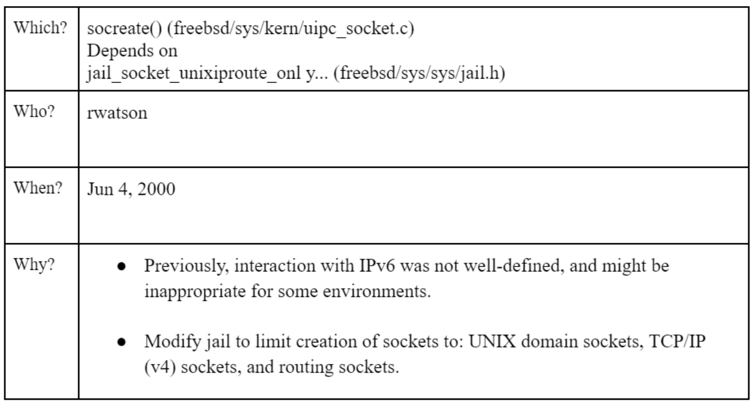
fildes[1] fildes[0]
process A
write end
fork
half-duplex UNIX pipe connecting two processes after a fork call and closing unused ends
read end
fildes[1] fildes[0]
process B

Figure 9: A diagram of the pipeline architecture in `POSIX`
[14]

In `POSIX`, `pipe()` takes an array of two integers. `fildes[1]` is used for writing, and `fildes[0]` is used for reading. In the above diagram, process B is forked from process A, and the written data from process A is fed through the pipe into process B. If it returns successfully, it writes file descriptors on each end of the pipeline. Otherwise, it returns `-1`.

5 Reflextion Analysis

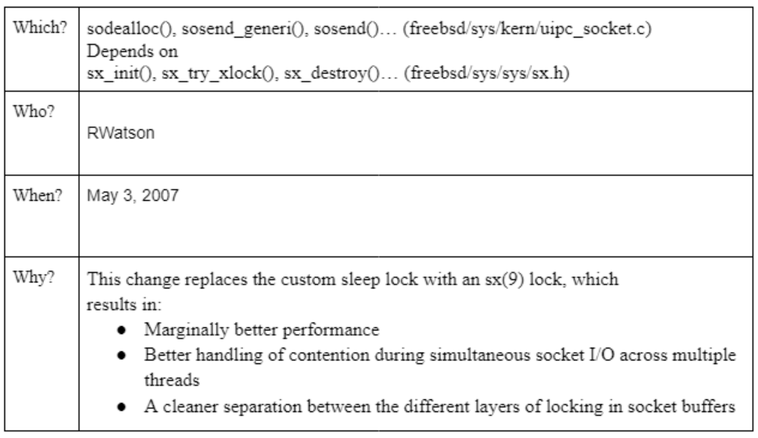
Now to start our reflection analysis of the introduced divergences from the conceptual architecture we will look at the dependencies from Inter-Process Communication to General System Interfaces. The General System Interfaces subsystem has a large amount of facilities and data structures used by IPC so we will only go over a couple. First, we will look at the dependency between `uipc_socket.c` and `jail.h`. The `uipc_socket.c` file implements a very large amount of socket facilities and is an extremely large file with many dependencies. The `jail.h` file implements the jail and prison data structures. This dependency was introduced because previously, interaction with IPv6 was not well-defined, and might be inappropriate for some environments. So what the author, rwatson, did to solve this was modify jail to limit creation of sockets to: UNIX domain sockets, TCP/IP (v4) sockets, and routing sockets. At the time of this change, this functionality was enabled by default, and toggleable using a variable in `jail.h`.



Which?	socreate() (freebsd/sys/kern/uipc_socket.c) Depends on jail_socket_unixiproute_onl y... (freebsd/sys/sys/jail.h)
Who?	rwatson
When?	Jun 4, 2000
Why?	• Previously, interaction with IPv6 was not well-defined, and might be inappropriate for some environments. • Modify jail to limit creation of sockets to: UNIX domain sockets, TCP/IP (v4) sockets, and routing sockets.

Figure 10: InterProcessCommunication ⟶ GeneralSystemInterfaces `jail.h` Reflection Analysis

This next dependency is between `uipc_socket.c` and `sx.h`; which implements the lock data structure / facilities. To give some context, the sx in `sx.h` stands for shared/exclusive and shared/exclusive locks are used to protect data that are read more often than they are written. The key difference between shared/exclusive locks and regular read-write locks is that they do not implement priority propagation like mutexes and reader/writer locks to prevent priority inversions. This change was also done by rwatson and was well documented. This change was an optimization which resulted in: Marginally better performance, Better handling of contention during simultaneous socket I/O across multiple threads, and A cleaner separation between the different layers of locking in socket buffers.



Which?	sodealloc(), sosend_generi(), sosend()… (freebsd/sys/kern/uipc_socket.c)
	Depends on
	sx_init(), sx_try_xlock(), sx_destroy()… (freebsd/sys/sys/sx.h)

Who?	RWatson
When?	May 3, 2007
Why?	This change replaces the custom sleep lock with an sx(9) lock, which results in: • Marginally better performance • Better handling of contention during simultaneous socket I/O across multiple threads • A cleaner separation between the different layers of locking in socket buffers

Figure 11: InterProcessCommunication ⟶ GeneralSystemInterfaces `sx.h` Reflection Analysis [10]

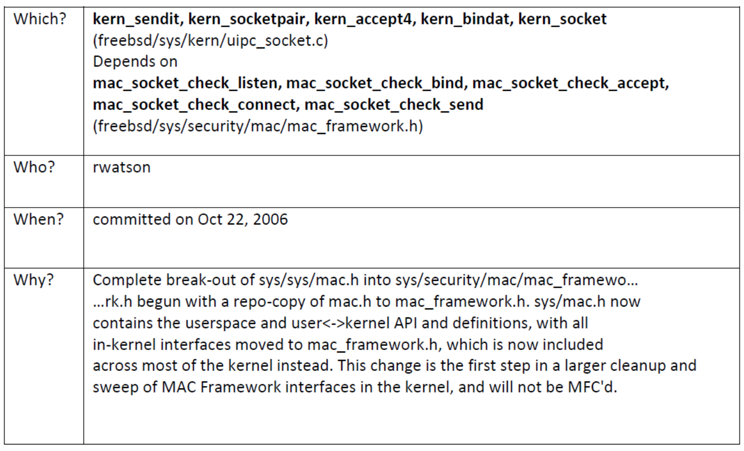
5.1 IPC dependence on the Security Subsystem

Another dependency that we discovered in the process of extracting the concrete architecture is that IPC depended on the Security subsystem. The Security subsystem in FreeBSD is responsible for enforcing security policies and mechanisms at various level of the system, including network security, user authentication, access control, and system integrity.

The IPC subsystem depends on the Security subsystem in two main ways:

5.1.1 Access Control

The IPC subsystem relies on the Security subsystem to enforce access control policies for IPC resources such as message queues, shared memory segments, and semaphores. The security subsystem provides mechanism such as file permissions and mandatory access control (MAC) frameworks to restrict access to IPC resources based on user identities, level of privilege the user has and security policies. We did a study of the source code FreeBSD and found the file level dependency that explains this relation of two subsystem:



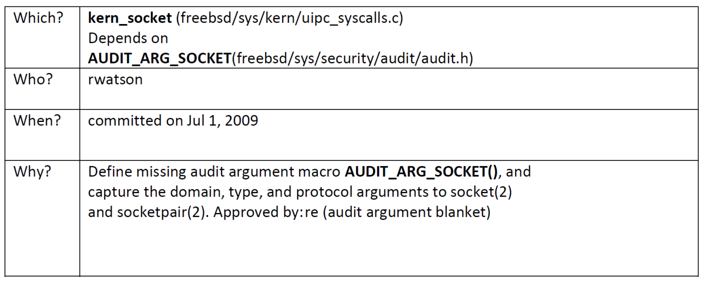
Which?	**kern_sendit, kern_socketpair, kern_accept4, kern_bindat, kern_socket** (freebsd/sys/kern/uipc_socket.c) Depends on **mac_socket_check_listen, mac_socket_check_bind, mac_socket_check_accept, mac_socket_check_connect, mac_socket_check_send** (freebsd/sys/security/mac/mac_framework.h)
Who?	rwatson
When?	committed on Oct 22, 2006
Why?	Complete break-out of sys/sys/mac.h into sys/security/mac/mac_framewo... ...rk.h begun with a repo-copy of mac.h to mac_framework.h. sys/mac.h now contains the userspace and user<->kernel API and definitions, with all in-kernel interfaces moved to mac_framework.h, which is now included across most of the kernel instead. This change is the first step in a larger cleanup and sweep of MAC Framework interfaces in the kernel, and will not be MFC'd.

Figure 12: InterProcessCommunication $\longrightarrow$ Security `mac_framework.h` Reflection Analysis

Here we see that a file named `uipc_socket.c` depends on a header file `mac_framework.h`, that is part of the security subsystem. `uipc_socket.c` contains implementation of socket related functions. This file is responsible for creating, managing, and interacting with sockets for network communication. the reason for the dependency to `mac_framework.h` is that it provides a mechanism for implementing and enforcing mandatory access control policies. Therefore, the inclusion of `mac_framework.h` is essential for ensuring the security of network communication in FreeBSD.

5.1.2 System Integrity

It is also essential that the integrity of IPC resources are maintained and that there is no unauthorized modification or tampering. The security subsystem provides mechanisms such for logging and monitoring system activity for this purpose. The auditing subsystem can track various types of system events, including process creation, IPC activity, and file access. We went ahead a tried to find this dependency in the source code and came across the following relation:



Which?	**kern_socket** (freebsd/sys/kern/uipc_syscalls.c) Depends on **AUDIT_ARG_SOCKET**(freebsd/sys/security/audit/audit.h)
Who?	rwatson
When?	committed on Jul 1, 2009
Why?	Define missing audit argument macro **AUDIT_ARG_SOCKET()**, and capture the domain, type, and protocol arguments to socket(2) and socketpair(2). Approved by:re (audit argument blanket)

Figure 13: InterProcessCommunication $\longrightarrow$ Security `audit.h` Reflection Analysis

The `audit.h` header file provides interface for the system's auditing subsystem, which is responsible for logging and monitoring system activity for security and compliance purposes. A file named `uipc_syscalls.c` that is part of the IPC subsystem depends on `audit.h` because it contains an important macro called `AUDIT_ARG_SOCKET()` [3]. This macro is used to capture the domain, type, and protocol arguments of `socket()` and `socketpair()` system calls. These arguments are important pieces of information for auditing network activity, as they indicate the type of socket that is being created. This way the OS gains the ability to log socket-related events to system audit trail [3]. This allows the system administrator to monitor network activity and detect and suspicious behavior [3].

6 Use Cases

6.1 Piping In Shell

The following sequence diagram shows how system calls can be used to pipe the output of one application to the input of another in FreeBSD, using a shell as an example program.

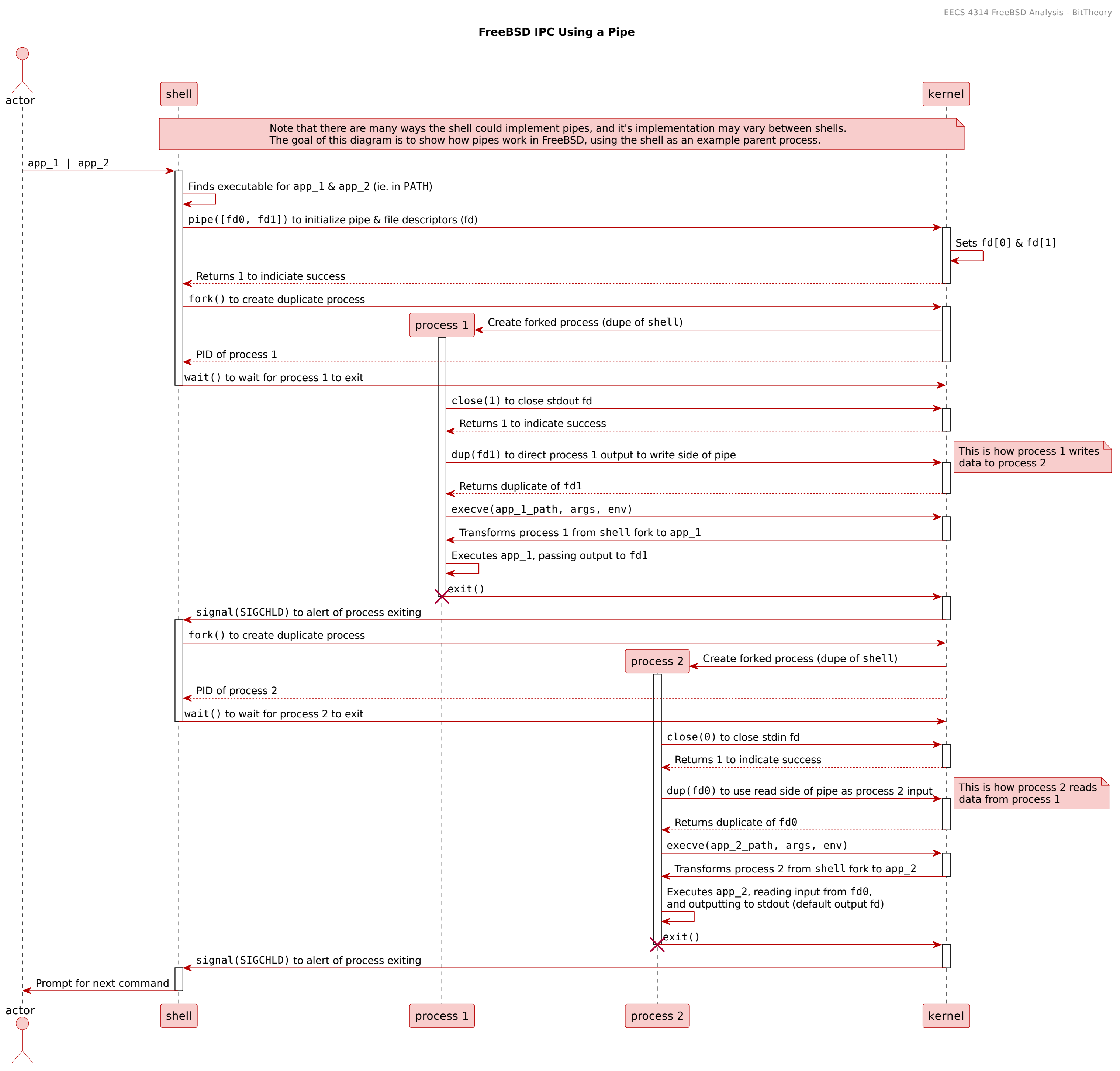

EECS 4314 FreeBSD Analysis - BitTheory
FreeBSD IPC Using a Pipe
actor
shell
kernel
Note that there are many ways the shell could implement pipes, and it's implementation may vary between shells.
The goal of this diagram is to show how pipes work in FreeBSD, using the shell as an example parent process.
app_1 | app_2
Finds executable for app_1 & app_2 (ie. in PATH)
pipe([fd0, fd1]) to initialize pipe & file descriptors (fd)
Sets fd[0] & fd[1]
Returns 1 to indiciate success
fork() to create duplicate process
process 1
Create forked process (dupe of shell)
PID of process 1
wait() to wait for process 1 to exit
close(1) to close stdout fd
Returns 1 to indicate success
dup(fd1) to direct process 1 output to write side of pipe
This is how process 1 writes data to process 2
Returns duplicate of fd1
execve(app_1_path, args, env)
Transforms process 1 from shell fork to app_1
Executes app_1, passing output to fd1
exit()
signal(SIGCHLD) to alert of process exiting
fork() to create duplicate process
process 2
Create forked process (dupe of shell)
PID of process 2
wait() to wait for process 2 to exit
close(0) to close stdin fd
Returns 1 to indicate success
dup(fd0) to use read side of pipe as process 2 input
This is how process 2 reads data from process 1
Returns duplicate of fd0
execve(app_2_path, args, env)
Transforms process 2 from shell fork to app_2
Executes app_2, reading input from fd0, and outputting to stdout (default output fd)
exit()
signal(SIGCHLD) to alert of process exiting
Prompt for next command
actor
shell
process 1
process 2
kernel

Figure 14: Sequence diagram showing pipes being used in the shell for IPC. [8][7][11][4][5][6][9][13]

7 Conclusion

In conclusion, the FreeBSD operating system has a powerful and flexible inter-process communication system which allows processes to communicate with each other, both through networks (Networking / Socket IPC) and within local computing environments (Single system / Local IPC).

On one hand, the Socket or networking IPC is a mechanism for communication between processes running on the same system or on different systems connected over a network. In this sense, data from the applications flows through a socket and through a networking layer. The received data, which also comes from the networking layer through the socket, then goes into the application. With sockets, not only do processes not have to run on the same machine in order to communicate between them, they do not even have to run under the same operating system.

Three main types of sockets are supported by the FreeBSD operating system: (1) stream sockets, which provide a connection-oriented, sequential, reliable and bi-directional communication channel; (2) datagram sockets, which provide a message-oriented communication channel in a connectionless manner; and (3) raw sockets, which allow direct access to the low level protocols of the network layer and facilitate sending/receiving network packets.

On the other hand, the single system or local IPC includes an integrated environment of semaphores, message queues, pipes and shared memory regions. Semaphores serve as a synchronization mechanism to allow different processes to access shared memory. Message queues, however, facilitate sending and receiving messages locally in a structured way. Along with pipes, they serve as a buffer that stores messages in a First-In-First-Out (FIFO) manner. These various mechanisms provide low-level access to regions of shared memory and sending/receiving messages between multiple processes.

Overall, FreeBSD provides a variety of IPC mechanisms that allow processes to communicate and coordinate with each other efficiently, whether on the same system or across a network.

8 Lessons Learned

After studying the concrete architecture of the IPC (Inter-Process Communication) system of FreeBSD, some of the lessons learned include:

- **Lesson 1:** The interprocess communication system of FreeBSD is designed to be modular, flexible and scalable, which means that different communication mechanisms between processes can be implemented. This provides a powerful IPC system for both local and networking environments.

- **Lesson 2:** As it includes mechanisms such as message queues or stream sockets, the FreeBSD's IPC system is designed to be reliable and to ensure structured bi-directional communication amongst various local or remote processes.

- **Lesson 3:** Sockets allow processes to communicate through a network, even within computers with different operating systems. They provide the means for application data to flow through the networking layer, be received by a different process on a different machine, and flow into the application of the new receiving environment.

- **Lesson 4:** Pipes are used in the shell for interprocess communication in a First-In-First-Out manner. It is a mechanism that allows process communication by sharing an unidirectional stream of data.

- **Lesson 5:** Overall, the power of the FreeBSD IPC subsystem provides structured, secure and reliable mechanisms for interprocess communication, making it a popular option being widely used in the industry.

9 Data Dictionary

- Understand: A source code extractor, used for extracting function relations
- jGrok: A tool, used for finding subsystem level dependencies

10 Naming Conventions

- IPC: Inter-Process Communication
- LSEdit: The Landscape Editor
- CFX: Source Code Extractor
- FIFO: First in First Out
- SX: Shared/Exclusive
- I/O: Input/Output

References

[1] Bell-Thomas, A. H. “Interprocess Communication in Freebsd 11: Performance Analysis.” DeepAI, University of Cambridge, 5 Aug. 2020, https://deepai.org/publication/interprocess-communication-in-freebsd-11-performance-analysis.

[2] “BitTheory source code.” README.md, https://github.com/BitTheoryProject/eecs4314-reports/tree/main/a2/src.

[3] “Chapter 18. Security Event Auditing.” FreeBSD Documentation Portal, https://docs.freebsd.org/en/-books/handbook/audit/.

[4] “FreeBSD Manual Pages.” Close(2), https://man.freebsd.org/cgi/man.cgi?query=close&sektion=2.

[5] “FreeBSD Manual Pages.” Dup(2), https://man.freebsd.org/cgi/man.cgi?query=dup&sek-tion=2&n=1.

[6] “FreeBSD Manual Pages.” Execve(2), https://man.freebsd.org/cgi/man.cgi?query=execve&sek-tion=2&n=1.

[7] “FreeBSD Manual Pages.” Fork(2), https://man.freebsd.org/cgi/man.cgi?fork(2).

[8] “FreeBSD Manual Pages.” Pipe(2), https://man.freebsd.org/cgi/man.cgi?pipe%282%29.

[9] “FreeBSD Manual Pages.” Signal(3), https://man.freebsd.org/cgi/man.cgi?sektion=3&query=signal.

[10] “FreeBSD Manual Pages.” SX(9), https://man.freebsd.org/cgi/man.cgi?sx(9).

[11] “FreeBSD Manual Pages.” Wait(2), https://man.freebsd.org/cgi/man.cgi?wait(2).

[12] McKusick, Marshall Kirk. Design and Implementation of the Freebsd Operating System. 2nd ed., Addison-Wesley Professional, 2015.

[13] Menon-Sen, Abhijit. “How Are Unix Pipes Implemented?” Toroid.org, 27 Mar. 2020, https://-toroid.org/unix-pipe-implementation.

[14] Rytarowski, Kamil. “Mastering Unix Pipes, Part 1.” Moritz Systems, 26 Nov. 2020, https://www.moritz.systems/blog/mastering-unix-pipes-part-1/.