university of groningen

faculty of science and engineering

Design and Implementation of a Software based Distributed Shared Memory System

Bachelor thesis

July 19, 2018

Student: Charles Randolph (s2897318)

Primary supervisor: Arnold Meijster

Secondary supervisor: Gerard Renardel de Lavalette

Contents

Abstract

Distributed Shared Memory (DSM) is a memory architecture which allows for addressing logically identical memory locations across physically separate memory spaces. The separation of memory space is defined as both policy based separation (enforced by the operating system's kernel) and actual physically separate machines. DSM architectures alleviate programmers from the task of micromanaging shared data, making it suitable for developing parallelized programs. In this paper, the design and implementation of a software based DSM for the Linux operating system is detailed and discussed.

Introduction

The objective of this paper is to detail the design choices and implementation specifics of a software based shared memory system for distributed computers. Distributed computer systems are informally defined as a network of physically separate machines working together concurrently in order to solve a common problem. The ability of such systems to offer both parallelism and concurrency makes them well suited for accelerating certain classes of computational tasks. Accordingly, distributed computer systems have many important scientific, business, and entertainment applications. The distributed shared memory (DSM) system detailed in this paper provides a mechanism for enabling programmers to parallelize their applications across such distributed systems. Although many DSM systems already exist, few meet the following desired requirements, prompting us to start anew: (1) The DSM system must be software based, and live entirely in userland. (2) The DSM system must run on the Linux kernel. (3) The DSM system must be easy to support and maintain. The remaining passages of the introduction will be used to establish a frame of reference for the reader; so to provide perspective to future decisions. In short order, the items to be addressed concern the existing paradigms for parallel computing, the motivations for designing a DSM, and the historical background of DSM research. In the coming text, the shorthand "DSM" and "DSMs" shall be used to refer to distributed shared memory system and its plural counterpart respectively.

The role of DSMs is best framed in the context of parallel computing at large. Historically, inter-process cooperation across distributed systems can be classified into one of the following two paradigms: message passing, and shared memory [4]. Message passing is a verbose and opaque approach to IPC which provides the programmer with facilities for exchanging information by means of sending and receiving messages. However, the task of parallelization itself remains up to the programmer. Shared memory, on the other hand, exchanges data transparently in the background between processes, necessitating that only a few synchronization primitives be provided. Both approaches have their specialized domains. Message passing schemes are ideal for loosely coupled distributed systems where memory models may not be exactly identical across different machines (e.g. a set of machines connected over a LAN). In contrast, shared memory models are better suited to tightly coupled multiprocessor systems where all processors maintain direct access to the same shared memory space. A distributed shared memory system attempts to extend the shared memory model by providing a virtual memory space between loosely coupled machines. It usually achieves this through a transparent message passing layer.

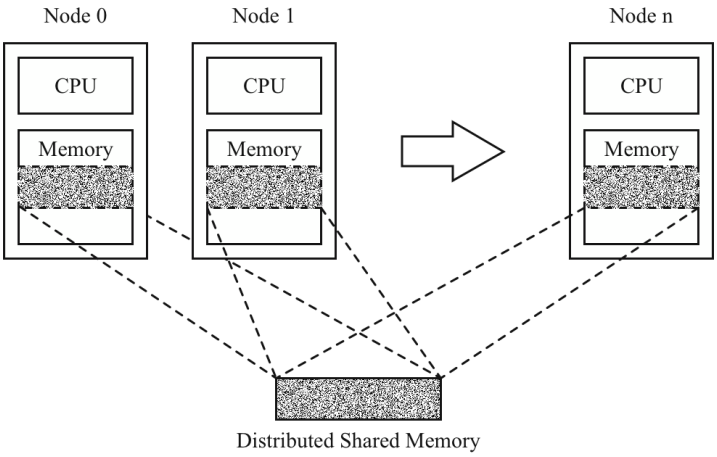

Figure 1.1: Illustration of distributed shared memory.

The decision to focus on a distributed shared memory system centers around the fact that it removes the burden of exchanging data from the programmer. This allows single threaded programs to be parallelized far easier than with message passing schemes. Converting a sequential, single threaded program to a parallelized one with a message passing interface often requires both an extensive rewrite, and that the programmer reformulates the problem conceptually. [2]. Furthermore, message passing requires a modest understanding of a particular library in order to be used effectively. This may make it less ideal than DSMs in scientific applications, where inexperienced programmers are burdened with integrating message passing semantics in concert with the conceptual problem.

Although distributed shared memory systems may appear as quite a niche subject in the field of computing science, it is in fact one that is neither new nor neglected. Research into DSM began as early as the 1960's, with rudimentary commercial solutions appearing in 1981 in the form of Apollo Computer's Aegis (Domain/OS) operating system [4]. Early products such as Domain/OS quickly spurred further research. A prime example is Yale's "IVY" DSM project, which appeared shortly after the appearance of Domain/OS, and ran a modified variant of the operating system. Throughout the rest of the decade, and well into the late 1990's, extensive amounts of time and resources were dedicated to the development of efficient designs and algorithms, culminating in a variety of commercial solutions and academic literature. However, despite the strong forays of the late 20st century into DSM and other forms of distributed computing, academic interest has largely faded away, and commercial solutions forgotten or deprecated. Ironically, the advances in computer hardware and networking infrastructure of the subsequent decade addressed many of the key problems plaguing early DSMs. Such problems included the 10-megabit per second bandwidth limit of Ethernet at the time, limited amounts of volatile memory, and extensive wait times for page replacements on hardware using physical disk stor-

age. The decision to focus on distributed shared memory systems is in some part influenced by the new opportunities offered by contemporary computer hardware.

Finally, the requirements of the distributed shared memory system were chosen carefully in order to maximize longevity and maintainability. The history of DSM design has born a plethora of solutions crafted on custom hardware architectures (MIT's ALEWIFE, BBN BUTTERFLY, and KSR1 to name a few) and kernel/operating system level modifications, many of which are now obsolete. Hardware based architectures – a popular choice at the time – rely on complex physical logic to drive down memory access speeds [4]. While such low level optimizations tend to offer the best performance, they are expensive, resistant to change, and in turn may be easily rendered obsolete. Similarly, software solutions such as METHER (which applies modifications to the SunOS kernel and filesystem), and LOCUST (which uses an extended version of the C language to provide it with compile-time dependency information), tie themselves to independent and highly sophisticated software systems, thus requiring extensive and often unsustainable upkeep. Consequentially, all kernel/operating system level modifications, custom languages and compilers, or hardware solutions were immediately off the table. In order to minimize portability issues, the Linux kernel was selected as the development platform for the project (though no modifications are made to it). Its open source nature and widespread popularity made it the natural candidate. In a similar vein, the C programming language was selected as the development language for the project.

The coming sections of this paper are structured in a top-down manner, so that theory precedes practice. This model is also recursively applied to the section structure where applicable. The intent is to present the reader with an abstract understanding of the problem and fill in the details later. Accordingly, the remainder of the paper begins with a summary of consistency models available to DSM designers, including that chosen for the project. Following that will come all subjects under the umbrella of the system architecture, beginning with an abstract component layout, extending to the memory model, and closing with the underlying message protocols and component behavior. Finally, the chief design challenges of the project are covered, and an evaluation of the DSM is presented. The text ends with a conclusion, including suggestions for future research .

Consistency Models

Each access to memory in an imperative computer program follows a total, strict order (i.e the last written value will be returned on the next read). These are the *coherence semantics* of a *strict consistency* model. Coherence semantics is a term that refers to the intended behavior of a distributed shared memory system with respect to load and store operations. [3] A consistency model is an implementation of coherence semantics. There are various types of coherence semantics, though strict consistency is the most rigid form. In practice, the strict consistency model is unfeasible in distributed systems due to the lack of a global clock. Therefore, most coherence semantics covered in this section are a reduced or "weakened" form, usually ordered through the use of a logical clock. Note that in this paper, the terms read and write will be used to refer to read and write *accesses* to memory.

2.1 Sequential Consistency

A *sequentially consistent* model is one that guarantees that the order in which a process executes events (both read and write operations) matches the order in which they are observed in the program. It however, only guarantees that *some order* will exist for events between different processes (though all will observe each others events in an ordered manner), and not what that exact ordering is. In other words, there is only a guarantee that local ordering is respected, and no guarantee for a single global ordering (see Figure 2.2). The memory consistency protocol must only ensure that it correctly interleaves reads and writes to shared memory atomically. This is generally done by serializing all events through a single node.

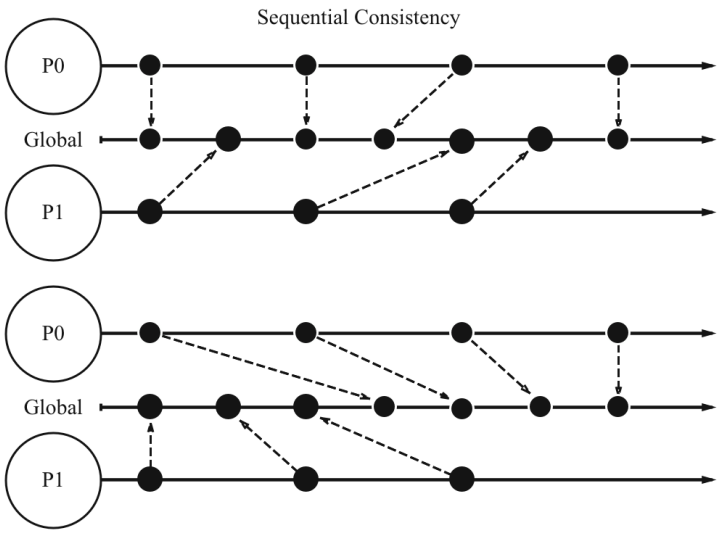

Figure 2.2: Two valid global load/store orderings for a sequentially consistent model

2.2 Processor Consistency

A *processor consistent* model is one that guarantees that *writes* issued by a given processor are never seen out of order to any other processor. Just like in sequentially consistent models, the total global ordering is not defined. However, unlike sequential consistency, processor consistent models allows read operations to bypass write operations, and does not guarantee atomicity in write operations. [3]

2.3 Causal Consistency

A distributed system is said to be *causally consistent* if there exists a partial order to the events of the system in agreement with Lamport's concept of "potential causality". A simple Lamport clock is a logical clock that defines the following *partial* relationship between two events: $a \rightarrow b \Rightarrow C(a) < C(b)$. In this context, the $\rightarrow$ symbol means "happened before". The relationship states that if event a happened before event b, then we can assert that the value of the clock when event a occurred is definitely less than the value of the clock when b occurred. However, the inverse relationship is meaningless: $C(a) < C(b) \not\Rightarrow a \rightarrow b$. The concept of an event a occurring before an event b is easily understandable in the context of a single process, where everything has a total order. However, it can also be extended to interprocess communication in the form of messages. A message from one process representing event a can only ever be received by another process as event b *after* it was sent, thus extending potential causality across processes. In this light, the lack of an inverse relationship also can be intuitively understood. If a process on which event a occurs never communicates with a process on which b occurs, then nothing can be said about the

relationship between a and b. The lack of a global clock means comparing the logical clock values of both events is meaningless. [9]

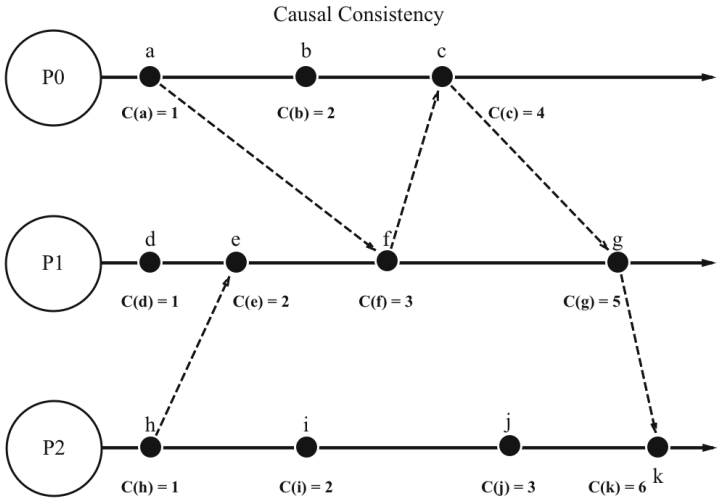

Figure 2.3: A Causal consistency model conforming to Lamport's potential causality

Figure 2.3 provides an illustration of a system applying causal relationships through the use of Lamport timestamps. Each process begins with an initial clock value of zero ($t_0 = 0$). This clock value is incremented before each event occurs, such that the time at $t_i = t_{i-1} + 1$. Whenever a message is sent to another process, it carries with it its own Lamport timestamp, that being the clock time when the sending event occurred. A process receiving a message obtains it with a clock time defined as: $max(t_{i-1}, t_{msg}) + 1$. This can be seen in Figure 2.3, where the clock time in P_0 jumps from 2 to 4 between events b and c. The partial order of relationships across processes may then be defined by the transitive closure property. Because $h \rightarrow e$, and $e \rightarrow f$ (all events within a single process have a strict ordering), and $f \rightarrow c$, then $h \rightarrow c$. This also means that $h \rightarrow k$, and $a \rightarrow k$. If a path can be found between events across processes, then there exists a partial ordering. However, nothing can be said about events i and b relative to each other, despite having the same local clock value. All such uncomparable events are known as *concurrent* events.

2.4 Weak Consistency

The final consistency model to be covered implements *weak consistency* semantics. In a system with weak consistency, the invocation of a synchronization operation causes the processor to stall until all previous accesses are globally performed. Accesses after the synchronization point must also wait for the global completion all previous synchronization accesses. Synchronization ac-

cesses themselves are guaranteed to be sequentially consistent. A weak consistency model effectively provides the user with the responsibility of implementing synchronization points. All other accesses must only obey local program order, but no global ordering is defined. [3]

2.5 Selected consistency model

A sequential consistency model was originally chosen for the DSM. However, this decision was quickly reversed, and a causal model selected instead. Although the simplicity of the sequential consistency model was attractive for an initial prototype, the fact that it necessitated *both* read and write operations be performed globally proved difficult to accommodate. The source of the problem is grounded in the design of the synchronization mechanism.

To understand how the synchronization mechanism works, a bit of background on virtual memory is useful. In most modern operating systems, a process has access to a virtual memory space, which the kernel transparently translates to physical memory space. Within this memory space, certain protections may be applied to units of memory, called pages. These protections control whether a process is allowed to access the page. Access violations are propagated to the process through a mechanism called signals. In the DSM, these signals are used to perform the synchronization. Because shared memory pages are protected against access operations, user attempts to access the page cause a violation, which the kernel then transforms into a signal and sends to the process. The synchronization itself occurs in special handlers attached to the reception of the violation signal.

While the functions attached to the reception of a signal are provided with some contextual information, such as the origin of the fault, they do not always know what specific type of access caused the violation. An access violation on memory protected against both reads and writes might be the result of an either an illegal read or write. The ambiguity is intolerable, given that specialized behavior must be taken depending on the attempted access. This, along with the potential performance implications that accompanied sending messages for read operations, rationalized the change.

System Architecture

The architecture of a DSM concerns not only the high level component layout, but also the memory model, choice of the messaging protocol, and behaviour of all constituent components. The following subsections will detail the design of the system in full, starting with its most abstract form.

3.1 The Big Picture

The DSM's abstract component layout is an apt starting point for building an understanding of the system. The primary objective of the DSM is to enable processes across different machines to cooperatively share memory during the execution of a single program. The term *session* was chosen to describe such an arrangement. In order for a session to take place, all participating processes must be able to coordinate a session's creation. To achieve this, a service must exist at a fixed location to ensure orderly startup. Figure 3.4 depicts a system that fulfills these requirements:

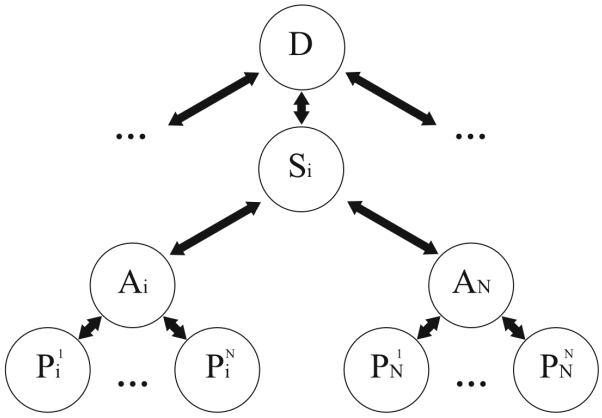

Figure 3.4: The abstract architecture of the distributed shared memory system.

Figure 3.4 features the following component types:

1. **D**: The session daemon.
2. **S**: A session server.
3. **A**: A local arbiter.
4. **P**: A local process.
5. **Arrow**: Connects communicating components.

The session daemon (D) fulfills the role of a fixed service. It must be started prior to any usage of the system, and is configured to listen for connections on a fixed port. Only one instance of a session daemon exists at any time, as illustrated in Figure 3.5. When the daemon receives a session request (Figure 3.5: 1), it creates a session by forking a session server (S). The session server facilitates a session by updating the daemon with its connection details (2). Once done, the session daemon can forward the details to requesting arbiters (A) (3). It also informs any waiting arbiters that made requests prior to the server's update message. A session server otherwise relays messages and performs functions in conjunction with the messaging protocol (4). Multiple session servers may exist at any time, though only one is shown in the diagram.

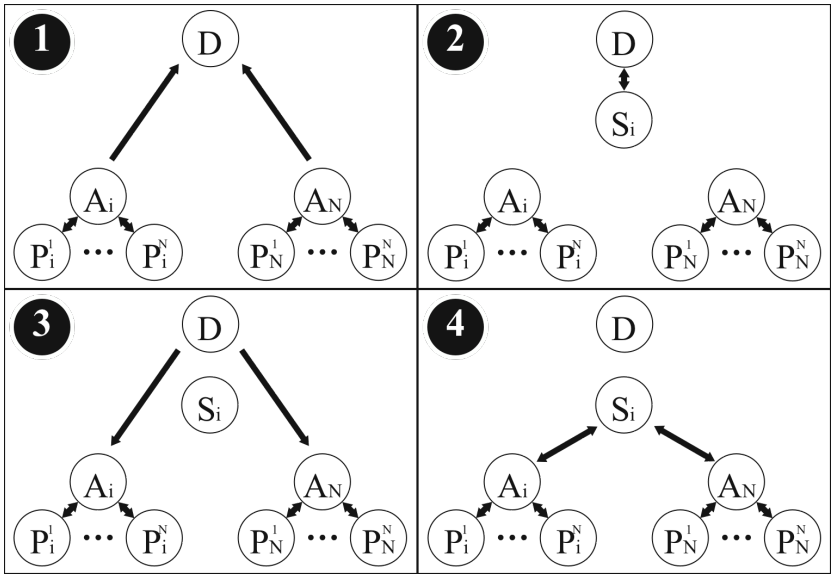

Figure 3.5: The session startup procedure.

The remaining two components consist of the local arbiter and process. A process is a just a standard UNIX process, and serves no special purpose. You may assume that all processes are participants by virtue of having made the necessary function calls available in the DSM's interface. The role of the arbiter carries more nuance though, as it initially appears as an unnecessary middle man between the server and the processes. The existence of the arbiter is justified

by the need to exert control over the processes of a machine when they're not performing write accesses. Given that a process will primarily be executing user written code, there exists no natural mechanism for assuming control between memory access violations. Such control is still desired though, for situations where the process must be paused in order to safely update the shared memory map. The arbiter provides this service by suspending and resuming processes on demand. It also performs other bookkeeping duties, such as creating and destroying shared resources. Only one arbiter instance exists per machine.

3.2 Memory Model

In the context of parallel systems, a memory model concerns the distribution of data across nodes. There are two primary methodologies for data distribution: replication, and partitioning. In a model employing replication, all nodes maintain a full copy of the shared memory space. Changes to memory in one node are typically reflected across all others. In contrast, a partitioning memory model may divide the shared memory space among the nodes, or supply it only on demand. The choice of a memory model is also intertwined with the coherence semantics of a system. For instance, a system with a sequential consistency model might be paired with a replicated memory scheme, owing to the fact that frequent global synchronizations would make shuffling partitions around expensive. [6]

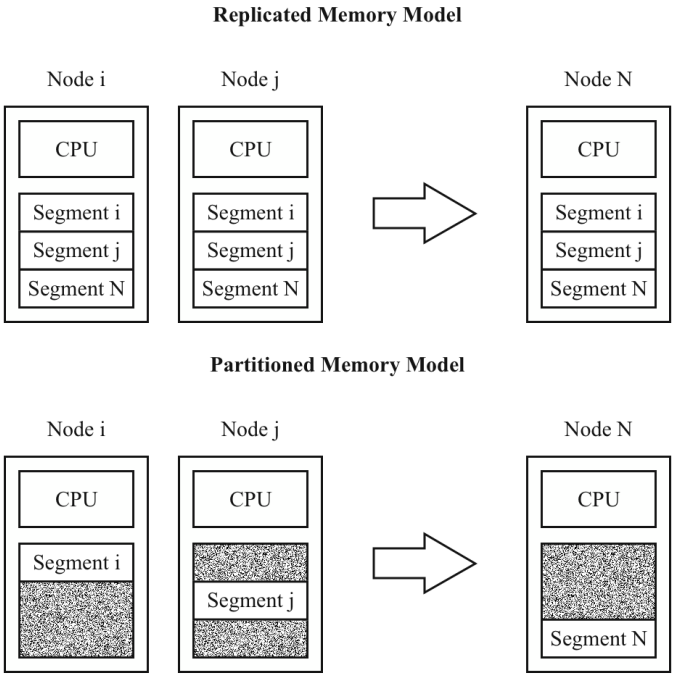

Figure 3.6: Replicated and Partitioned Memory Models

3.2.1 POSIX Shared Memory

The chosen memory model for the DSM is replication. Despite the more sophisticated behaviours supported by partitioned models, the sheer simplicity of replication made it a more compelling choice. In practice, replication is achieved through a combination of process forks and POSIX shared memory maps. POSIX shared memory is a mechanism that allows memory to be shared between unrelated processes, usually in the form of a mapped file. It is available on most UNIX like systems, including current versions of the Linux kernel. Multiple processes on a single system may write and read from the shared memory space, which is maintained by the kernel. Processes may additionally ascribe individual memory protections to the shared region, a feature indispensable to the project's success. However, the user is still responsible for avoiding race conditions and enforcing atomic access to critical resources. Finally, the shared maps have kernel persistence, meaning they exist until explicitly deleted or a reboot occurs. [11]

3.2.2 Implementation Details

In the DSM, shared memory maps are created by a local arbiter on initialization. When n processes run their initializers simultaneously, all proceed to fork an arbiter. Because an arbiter attempts to bind and listen to a fixed port, only one succeeds. The arbiter that successfully binds to the port is also that which creates the POSIX shared map as a file. After successfully connecting to the arbiter, the remaining (normal) processes map the shared file into memory. This mechanism ensures that (1) The shared map exists before all standard processes attempt to map it. (2) Only one shared map is created.

Destruction of shared memory is split into two stages. The first stage concerns the mapped process memory, and the second concerns the persistent shared memory object. Whenever a process dies, the shared memory map is automatically removed by the kernel. However, the destructor function in the DSM interface performs this task manually. This ensures that a process can participate in sequential sessions without running out of memory. The persistent shared memory object is removed by a special daemon who's sole task is to wait for the arbiter to terminate. When the arbiter does terminate, be it quietly or because of an error, the daemon always succeeds in removing the shared object.

3.3 Messaging Protocol

The causal consistency model detailed in Section 2.5 is implemented in the form of the DSM's message passing protocol. The message passing protocol is designed to support the following operations, each of which will be addressed independently:

1. Initialization and destruction of a session server.
2. Execution of a global write.
3. Waiting on a synchronization barrier.

4. Posting or waiting on a named semaphore.

5. Obtaining a process global identifier.

An operation usually involves an exchange of messages between two or three parties. Messages in the DSM are described by a type and accompanying payload, though not all messages need payloads. Message payloads vary by type, but the total message size is always fixed at 64 bytes. This message size fits the largest payload with some space to spare. The definition of the message type is given below, though the payload definitions are too lengthy to fit.

```
typedef struct dsm_msg {
    dsm_msg_t type;             // Message Type.
    union {
     dsm_payload_sid sid;    // Session data.
     dsm_payload_proc proc;  // Process data.
     dsm_payload_task task;  // Task data.
     dsm_payload_data data;  // Update data.
     dsm_payload_sem sem;    // Semaphore data.
    };
} dsm_msg;
```

Figure 3.7: The message type definition (located in file dsm_msg.h).

The DSM employs custom packing routines to exchange messages over a network. The term *marshalling* describes the act of transforming data from host form to network form, and the term *unmarshalling* describes the inverse operation. Because the C programming language does not make any guarantees concerning how padding bytes are added to structures in memory, it is unsafe to both send and expect data to match the locally computed size of the message type (`sizeof(dsm_msg)`). [12] Marshalling and unmarshalling routines eliminate this source of error by packing the structure instance into a compact fixed length byte-string before transmission, and re-assigning the structure values after transmission. Although it is expected that processing nodes share the same memory model, the same cannot be said for the session-server or daemon. Neither are required to read or install memory content sent between nodes. Finally, the protocol chosen for message transmission is TCP. The redundancy offered by the protocol was considered more valuable for stability purposes during development than the speed increase that alternatives like UDP would offer. [8]

3.3.1 Session Initialization and Destruction

Session initialization is touched upon briefly in the Section 3.1, and is illustrated in Figure 3.5. The initialization and destruction procedures involves the following metadata:

Message Type	Definition	Payload
GET_SID	Request session contact details.	char[32], int32
SET_SID	Update session contact details.	
DEL_SID	Remove session contact details.	

The common message payload used supports a 32 byte character string named `sid_name`, and a union integer type which can either be assigned by field name `nproc` or field name `port`. During initialization, the `sid_name` field is filled with an identifier specifying a shared session to be joined or created. The `nproc` field is set with the *total* number of processes expected to participate in the shared session. The message is dispatched to the session daemon by the arbiter. Once received by the session daemon, two actions are taken:

1. If the session identifier does not exist in the daemon's session table, a table entry is created, and the port of the associated server set to an invalid value. The session server is then forked with both the `sid_name` and `nproc` values as input, and the requesting arbiter is pushed to a pollable list. It is also tagged with an ID linking it to the session.

2. If the session identifier does exist, then the port is looked up by the session daemon. If it is invalid, the requesting arbiter is tagged with an ID linking it to the session, and then pushed to a list of pollable (active) connections called the *pollable list*. If the port is set, the message type is switched to `DSM_MSG_SET_SID`, and the `port` field of the payload filled in with the server port. The message is then dispatched to the arbiter, and the connection closed.

The daemon retains all arbiters that have requested a session server if the server has yet to check in. The server checks in by dispatching a `SET_SID` message to the daemon when ready, and sets the `port` field to whatever free port it was assigned. When received by the session daemon, the table entry for the server is updated with the new port. The list of pollable arbiters is then scanned, and any arbiters waiting on that particular session are send the same message bearing the identifier and port. Then, they are disconnected. Once the shared memory session has ended, the session server dispatches a message of type `DEL_SID` to the server, setting only its session identifier. The session is then looked up in the table, and the entry removed. A final scan of the pollable arbiters is made, and any arbiters erroneously waiting on the session are disconnected and removed.

3.3.2 Global Writes

Write operations are by far the most complicated operation supported by the message protocol. It involves no fewer than five steps, and requires a large set of different messages. The server plays a crucial role in the global synchronization operation, and will be covered in more depth later on. This subsection will focus chiefly on the messages exchanged.

Message Type	Definition	Payload
CNT_ALL	Continue all local processes.	None
GOT_DATA	Data acquisition acknowledgment.	int32
WRT_NOW	Instructs process to begin write.	
REQ_WRT	Request to perform a write.	
WRT_DATA	Write metadata (offset, size, bytes).	int32[2], byte[8]

The Write Request

Write accesses are captured by a specialized piece of software that is described in more depth later. When an access attempt is captured, the process that wishes to write must inform the system of its intent. A message of type `REQ_WRT` is then sent, with the integer payload value set to the process's PID. The arbiter receives this request, and marks the process as queued in a special table. It then forwards it to the session server. If another write is in progress, the request is queued as a unique pairing of the arbiter's file-descriptor (socket) and the requesting process's PID. If no write is in progress, the session server begins a new synchronization operation. Meanwhile, the process that has issued a request is blocked waiting for a go-ahead.

Synchronization Exchange

The go-ahead for performing a write begins when an arbiter receives a message of type `WRT_NOW` from the server with the PID of the process-to-write in the payload. The arbiter forwards this message to the waiting process by looking up its socket in a special table. The recipient process, previously blocked waiting to receive the acknowledgement, unblocks and performs the write. Once the write is complete, special handlers construct a message of type `WRT_DATA`. In it, the offset of the written memory in the shared map space is computed and installed, and a sequence of eight bytes is copied into the message payload. The copied sequence begins at the address in memory that the former attempt at writing had caused the process to block at. The specification of eight bytes seems peculiar, but is chosen because eight bytes also represents the widest type that fits an Intel x86-64 general purpose register. Because the system synchronizes on an instruction-by-instruction basis, the largest amount of data that could have been written to the shared memory map must match the largest type that fits in the registers. The message is then forwarded to the arbiter.

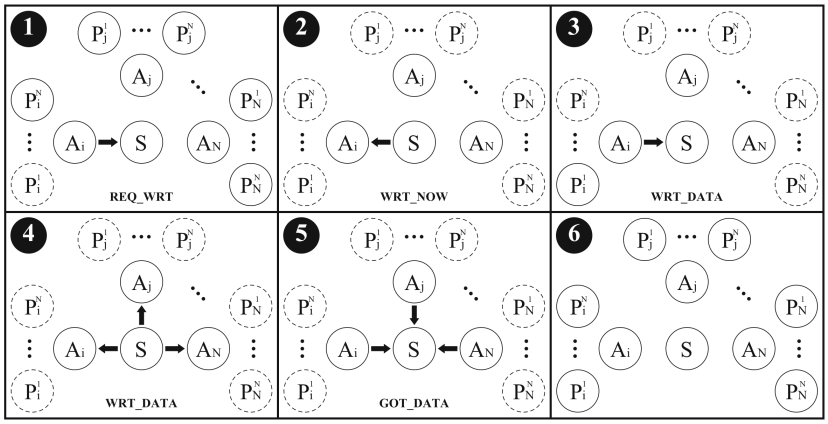

Figure 3.8: An illustration of the synchronization procedure for write operations.

An arbiter receiving a message of type `WRT_DATA` takes two different actions depending on the message sender. If the sender is the session-server, it must be data to apply to its local shared map. Otherwise, the message originates from inside the machine, and is simply forwarded to the session-server. In the case of the former, the arbiter temporarily lifts protections to the shared memory map, and applies the update, then restores protections. Regardless of who sent the message, the arbiter always sends an acknowledgement to the server that it received message of type `WRT_DATA`. The acknowledgment message has type `GOT_DATA`, and the number of local processes is attached as the payload.

Finally, the session-server reacts to `WRT_DATA` by forwarding the message to all connected arbiters. It then waits to receive messages of type `GOT_DATA`. For each message of that type it receives, it adds the attached process count to a running sum. When the total number of registered processes are all accounted for, it checks the write-queue to see if any more writes need to be done. If so, it dispatches another go-ahead message to the arbiter of the next writer and continues from there (i.e jumps to step two from step five in Figure 3.8). Otherwise, the global write is complete.

3.3.3 Synchronization Barriers

Although the core attraction of a DSM is its ability to alleviate the user from the burden of explicit synchronization, no proper DSM is complete without a few essential synchronization primitives. One such primitive is the barrier. A barrier is a tool that allows all processes to align themselves at a certain point in program execution. Barriers effectively prevent all further accesses until all participating processes have reached it. To use a barrier, one must embed a directive in the program. They are typically embedded after naturally occurring points of data convergence in programs. For instance, a sum computed over a set of nodes must use a barrier before the total is printed on output. If this is not done, the outputted sum may not reflect the true count. This would occur because the process printing the output finished before all processes had a chance to contribute. [1]

Message Type	Definition	Payload
HIT_BAR	Request session contact details.	int32
REL_BAR	Update session contact details.	None

In the DSM interface provided to users, the barrier manifests itself as a function call. When invoked, the calling process dispatches a message of type HIT_BAR, and sets the integer payload value to that of its PID. The process then suspends itself with a SIGTSTP signal. When the dispatched message is received by the arbiter, it marks the process as blocked in the table, and forwards the message to the server. The server then increments a counter of blocked processes. When the counter reaches the number of registered processes, the server dispatches a message of type REL_BAR with no payload, and resets the counter to zero. All arbiters receiving the message send SIGCONT signals to local processes marked as blocked. This resumes the execution of those processes, and completes the function of the barrier.

3.3.4 Semaphore Operations

Semaphores are the other synchronization primitive supported by the messaging protocol. A semaphore is a special construct that enables a programmer to easily build critical sections into their program. Doing so provides mutual exclusion in situations where barriers would not suffice. As a simple example, consider a program where each process writes data to a file. Without access control, several processes might write to the output at once. Although they will be multiplexed by the operating system scheduler, the output will still risk being garbled. A semaphore can indirectly provide mutually exclusive access to the output resource by allowing only one process to "possess" it at a time. Other processes that attempt to take the semaphore are suspended until the owner releases it.

Message Type	Definition	Payload
POST_SEM	Decrements named semaphore value.	int32, char[32]
WAIT_SEM	Increments named semaphore value.	

To improve clarity, this subsection is divided into two segments. Once concerns the abstract behaviour of semaphores, and the other the implementation specifics.

Semaphore Behaviour

A semaphore is a synchronization primitive with an integer value and two supported operations. One operation, called a post or "up", atomically increments the value of the semaphore. The other operation, called a wait or "down", atomically decrements the value. A semaphore with a natural number value may be decremented or incremented without any side effects. However, a semaphore

with a value of zero cannot be decremented. Doing so blocks the caller, meaning they are suspended and unable to "deposit" their "down". Multiple callers, typically processes, may wait in this blocked state. Incrementing a semaphore with a value of zero can result in one of two actions being performed. If no process is waiting to "deposit" their "down", the semaphore value is simply incremented. Otherwise, the value remains zero, but one (of potentially several) blocked process is unblocked. Finally, because incrementing a semaphore has no direct consequence to the caller, it is always implemented as a non-blocking call. [5]

Semaphore Implementation

In the DSM, a semaphore is an entity identified by a 32-byte character string, and initialized by default with a value of one. No initializers for semaphores exist in the interface. Instead, they are initialized on the fly if the identifer does not exist in the table. A process that wishes to wait on or "down" a semaphore does so by executing a function call listed in the DSM's interface. The function call takes an identifier, which is then placed in the payload of a message of type `WAIT_SEM`. The PID of the calling process is also attached to the payload, and then the message is sent to the arbiter. The function call is blocking, meaning that the caller blocks waiting to an acknowledgement that the "down" operation succeeded. When received by the arbiter, the process is marked as blocked, and the message is forwarded to the session-server. The session server then takes one of two actions.

1. If the named semaphore has a positive nonzero value, it is decremented, and an acknowledgment is sent back in the form of a message bearing type `POST_SEM` with the caller's PID in the payload.

2. Otherwise, the process is marked in a table as blocked, and the semaphore's internal ID is attached to the process entry. No acknowledgement is sent in this case.

If the arbiter receives an acknowledgement, it looks up the socket of the blocked process using the PID in the message payload, and forwards it to the blocked process. The blocked process may then continue.

Posting to a semaphore also involves a call to an interface function. This function constructs a message of type `POST_SEM`, and attaches the identifier supplied in the parameters to the message payload. Unlike the wait message, no PID is needed here. The message is then dispatched to the local arbiter, which forwards it immediately to the session server. The function is also non-blocking, meaning it returns immediately. Once received by the session-server, one of two actions may occur.

1. If the semaphore has a positive nonzero value, then it is simply incremented and nothing happens.

2. If the semaphore has a value of zero, then the process table is searched for any blocked processes that are linked to that named semaphore. If one is found, it is sent an unblock message and the semaphore value remains zero. Otherwise, the value is simply incremented.

Finally, the DSM does not yet support the destruction of semaphores. This may be performed in future work.

3.3.5 Global Identifiers

The final mechanism to be covered is that of global identifiers. A global identifier allows a processing primitive to distinguish itself from other members of a group of processing primitives. In this DSM, a processing primitive is the standard process. Although processes on a machine already have a mechanism for identifying themselves locally (PID), the DSM must provide a new identifier for distributed systems where more than one local process might share the same PID. Most parallel programming libraries provide a facility like this, where the identifier is typically an integer. This same approach is used by the DSM. [13]

Message Type	Definition	Payload
`ADD_PID`	Registers process.	int32, int32
`SET_GID`	Sets process global identifier.	

When a process establishes a connection to its local arbiter, it first dispatches a message of type `ADD_PID`, and copies the value of its local PID to the integer field `pid` in the message payload. When this message is received by the arbiter, the PID is read from the `pid` field and logged in a process table. The arbiter then forwards the message to the session server. The session server is the only entity that has the means to prescribe global identifiers. It does so with a simple integer counter. Each time it receives a forwarded `ADD_PID` message, it pairs the value of the counter to the PID in the payload, writes the PID and GID into a table, and increments the counter. A message is then sent back to the arbiter. This message is of type `SET_GID`, and contains (1) the PID of the process to whom the GID belongs and (2) the GID itself.

Although one might imagine that the arbiter simply forwards all `SET_GID` messages back to the processes as they arrive, it actually holds onto them. Newly arrived `SET_GID` messages are simply used to update the arbiter's process table with the new GIDs. What the arbiter is really waiting for is a message from the server of type `CNT_ALL`. This message is dispatched when all processes have sent an `ADD_PID` message to the server. It marks the start of the shared session. Only after receiving this message does the arbiter dispatch messages of type `SET_GID` to each process with its GID. Because the processes were blocked waiting for the return message, they now unblock, set their GID locally, and begin running the user program. A function provided in the DSM's interface then allows users to retrieve the GID.

3.4 Component Behaviour

The implemented DSM contains many different components. In previous sections, most of these were introduced and discussed with respect to their role's in certain system mechanisms. However, there remains to be a comprehensive breakdown of each individual component. Unfortunately, writing an individual

breakdown of all components would be lengthy and tedious. Therefore, only the most important components will be discussed in this section.

3.4.1 The Session Server

The session server is the central hub of a shared memory session, in all senses of the word. From a literal standpoint, the session server forms the center of a topographical star network, with all arbiters connected to it. In a more metaphorical sense, the session server performs virtually all the services that the DSM offers. It orchestrates global accesses, maintains the named semaphores, establishes barriers, and assigns global identifiers. The server must also be flexible enough to tolerate unexpected messages, concurrent write requests, and the other idiosyncrasies of networked systems. The breakdown of the session server will begin with its abstract behaviour model, then move to cover the underlying data structures and logic.

Behaviour Model

The session server is designed to marry two specific mechanics together. The first mechanic is technical, the second behavioural. From a technical standpoint, the server must be able to always respond to both new connections and messages. This is essential for avoiding socket timeouts, and to prevent socket buffers from overflowing as messages pile up. From a behavioural standpoint, the server must be able to respond to messages from arbiters, even in the midst of different activities. As an example, consider a process that performs a wait on a named semaphore, and is blocked waiting for the acknowledgment. Let's also suppose a barrier is being enforced by the server. If the server is programmed such that it must complete the barrier before serving other requests, a deadlock may occur. In the example, this would happen because the blocked process can't reach the barrier until it gets the acknowledgment, and it won't get the acknowledgement until the server can drop the barrier.

The example demonstrates how the server, and indeed all message switching components of the DSM, must be able to carry out activities in incremental steps. This is achieved by mapping functions to messages. Because different messages correspond to different states, all services performed by the server are message driven, and the state of a component is updated in response to messages.

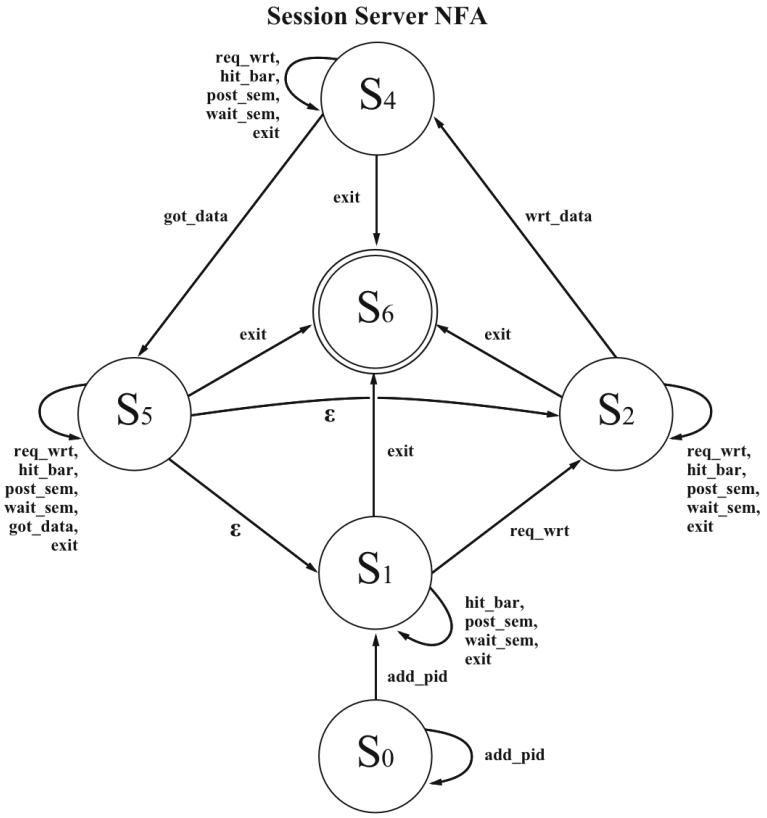

Figure 3.9: A non-deterministic finite automaton representing the session-server.

The non-deterministic finite automaton (NFA) depicted in Figure 3.9 describes the general behaviour of the session server, with the arcs between different states labeled with corresponding messages. For instance, the transition from S_0 to S_1 is the result of the final `ADD_PID` message that kicks off the start of the session. On a larger scale, the rhombus-like message loop represents how the server proceeds through the stages of a global access. The self arcs demonstrate how no matter what stage of a global access the session-server is in, it can still respond to other events. Messages that aren't accepted in various states are enforced in the implementation by assertion checks. The design of the server is far from perfect though. The session-server may always accept `EXIT` messages, even if it might cause a deadlock (the server might wait infinitely for all processes to be accounted for).

Data Structures

A large assortment of data structures are used in the server, each of which will be briefly described.

1. **Pollable Set**: The pollable set is a structure containing a 1D array of type `struct pollfd` for use with the library `poll` call. The data structure acts as a wrapper; automatically removing, shuffling, and resizing the array as needed. The `poll` library call allows the program to sleep on an array of sockets wrapped in type `struct pollfd`. It awakens whenever new data is available on any socket, making it ideal for reacting to messages from a collection of connected arbiters.

2. **Function Map**: The function map is a static 1D array. It is at least as large as the number of unique messages in the messaging protocol. Each message type in the protocol has an integer value. This integer value serves as an index into the function map, which is an array of function pointers. Therefore, whenever a message is received, it is simply decoded, and then the appropriate handler function is found and called with the message as input.

3. **Operation Queue**: The operation queue stores write-requests. It presents a high level queue interface.

4. **Process Table**: The process table is an array of linked lists, one list for each connected arbiter. The linked lists contains entries listing processes associated with those arbiters. The process table tracks the state of each process, and includes other useful information, such as which semaphore a process might be blocked on.

5. **Semaphore Hash Table**: The semaphore hash table is a wrapper for a generic hash table interface used throughout the entire DSM. It stores semaphores of type `dsm_sem_t`.

3.4.2 The Local Arbiter

The local arbiter performs the task of managing all local processes. It switches messages, ensures orderly startup, updates and tracks the state of processes, and dispatches signals. To a certain extent, it also simplifies the system architecture by providing a high level interface to the session server.

Behaviour Model

Unlike the session-server, the local arbiter does not transition between states often, save for the initialization phase. However, most of its complexity is not accurately captured by a state transition diagram.

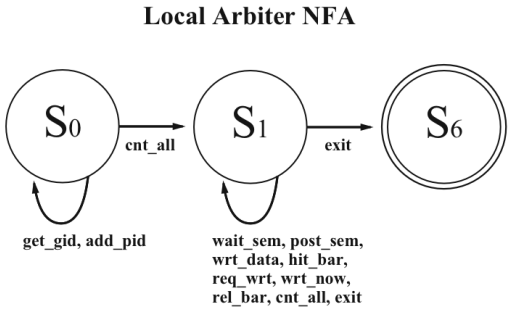

Figure 3.10: A non-deterministic finite automaton representing the local arbiter.

A large amount of the arbiter's logic involves assessing the state of local processes. The arbiter tracks whether processes are queued, blocked, or stopped. It administrates the start of a session locally, and switches messages or builds new ones. Much like the session server, the arbiter also routinely validates messages in order to minimize undesired behaviour. In short, the arbiter performs a lot of the bookkeeping tasks that would encumber the session server. To maintain simplicity, the arbiter's bookkeeping logic is largely split up into functions that can be mapped to the process table. This allows the bulk of the logic to be decoupled from the message handling routines.

Data Structures

The local arbiter uses many of the same data structures found in the session-server. The same goes for the session daemon (not shown). Special care was taken in the project to produce reusable data structures.

1. **Pollable Set**: The pollable set performs exactly the same task as that described in Section 3.4.1.

2. **Function Map**: The function map performs the same task as that described in Section 3.4.1.

3. **Process Table**: The process table described in Section 3.4.1 is the same one used by the local arbiter. However, the local arbiter is the only one of these components that actually makes full use of the table. The session-server does not track process state, other than whether or not a process is blocked on a semaphore.

Design Challenges

The defining feature of distributed shared memory is its ability to capture memory accesses transparently. Indeed, this is perhaps the only quality that truly severs it from message passing. As of this point, the reader has been exposed to the subtleties of consistency models, the details of the messaging protocol, and the inner plumbing of the system. In short, the big picture has been painted. Yet, little has been made of what should be the system's defining mechanism: capturing memory accesses. The task of developing this mechanism was the project's very first endeavour, and formed the foundation upon which the rest of the system was built.

To understand the problems that entail capturing write operations in a process, a certain familiarity of operating systems is needed. A core task of any operating system is to run programs. A process, as seen frequently throughout this paper, is effectively a program in execution. Processes may be described in some sense as a special environment in which a program is run. The operating system provides the process with a unique virtual address space, a runtime stack for functions, and facilities for allocating memory, manipulating files, and communicating with other processes. The operating system also decides when to schedule the process on the CPU, and when to put the process to sleep – or wake it up – as needed. Enacting changes to memory, as one might have gathered by now, is also an operating system responsibility.
[10]

So what makes capturing accesses so hard? The answer is tied to fact that we wish to do so *transparently*. When a user writes a program to be run on an operating system, they are not required to implement the sophisticated logic described earlier each time. A high level assignment to a memory address just works, and no more thought need be put to it. Likewise, we cannot require the user to perform any additional work for the DSM. The access must be captured invisibly. Because the operating system is responsible for carrying out access logic, it's easy to understand why tweaking it to capture the access at the kernel level is popular. It may not be trivial to perform, but it is elegant. Unfortunately, as elaborated upon in the introduction, the elegance comes at a maintainability cost we aren't willing to pay.

Ultimately, the problem of capturing accesses was distilled to two specific sub-problems. (1) How can we get the kernel to inform us that an access is occurring. (2) How can we capture the change performed by the access?

4.1 Signals

The problem of knowing when an access has occurred is solved by taking advantage of signals. Signals are a mechanism for performing "software interrupts" on UNIX-like systems. They may be issued by the kernel itself, or sent explicitly between processes in a crude form of IPC. For example, when a user pauses a process with a key-combination, the kernel enacts the suspension by delivering a signal to the process. In a similar manner, a process that divides by zero, or accesses memory out of bounds, is sent a tailored signal. [7] Of more interest, however, is the ability to register handler functions to signals. A handler allows the user to define the behaviour on arrival of a signal, and can be useful in providing advanced error handling or hooking in special functionality.

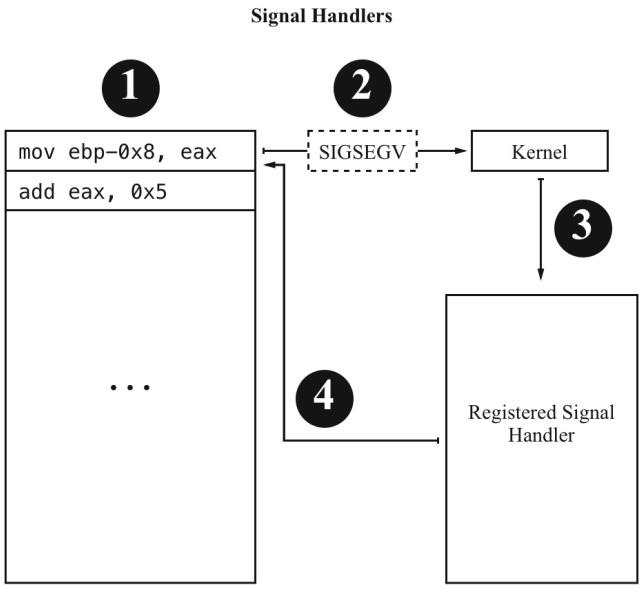

Figure 4.11: Signal handler invocation.

Signal Handler Steps

When an program instruction (1) triggers a signal (2), the kernel checks to see if any signal handlers are installed for that particular signal. If so, it invokes the handler (3). Once the handler is finished, the kernel will typically *restart* the last instruction (4). However, if the source of the fault has not changed, the signal will be raised again.

In order to know when a memory access was occurring, a handler was written to capture `SIGSEGV`. `SIGSEGV` is a signal sent when the program attempts to access memory it shouldn't. The idea was that if a segment of the process heap memory could be protected, then by attaching the `SIGSEGV` handler, we could capture the accesses to that segment transparently.

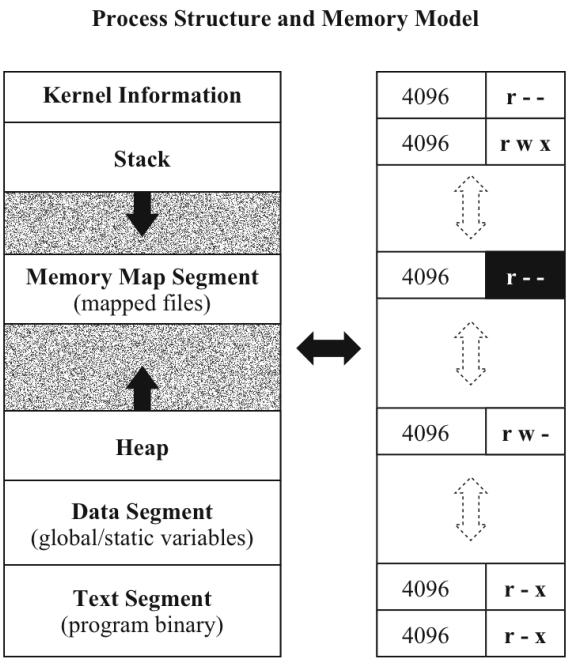

Figure 4.12: Illustration of protected heap memory.

Protected memory

The illustration depicts a mapping between the layout of a process and the underlying memory structure. Each memory page – the smallest continuous block of memory supported by the operating system – has a set of protection bits associated with it. By purposefully setting read-only protections to a specific memory page on the heap, attempts to write will trigger a segmentation fault (`SIGSEGV`).

The attempt to capture write operations using a `SIGSEGV` handler initially proved promising. When registering a signal handler, a decision may be made to receive extra information concerning the signal. This information varies depending on the signal type, but in the case of `SIGSEGV`, it provides both the program counter and the address of the accessed memory which caused the fault. With this, we know where the user is trying to write. However, there remains two crucial problems. The first was that the signal handler interrupts *before* the write is completed. This makes a lot of sense of course. What good would memory protections be if the kernel only captured illegal accesses after they had been performed? Nonetheless, it meant there was no way to know what would be written. The second problem was that the source of the signal had to be resolved if the program was to proceed. In this case, it meant granting write access to the protected memory page. Doing so, however, meant that further accesses to the page would no longer trigger the `SIGSEGV` signal. There was no way to grant access "just once".

4.2 Instruction Injection

In order to capture accesses, a rather extreme workaround was proposed. When the `SIGSEGV` handler was fired, the *next* instruction was to be cut out and replaced with a fault. This meant that the process could be granted permission to modify the protected memory, knowing that on the next instruction, the fault would raise a signal which could be captured by a signal handler. Within the second signal handler, the bad instruction would be replaced by the original content, the change synchronized (using the previously stored address of affected memory), and protections to the memory space re-applied. This all happens in the span of just two instructions, ensuring that changes are captured atomically.

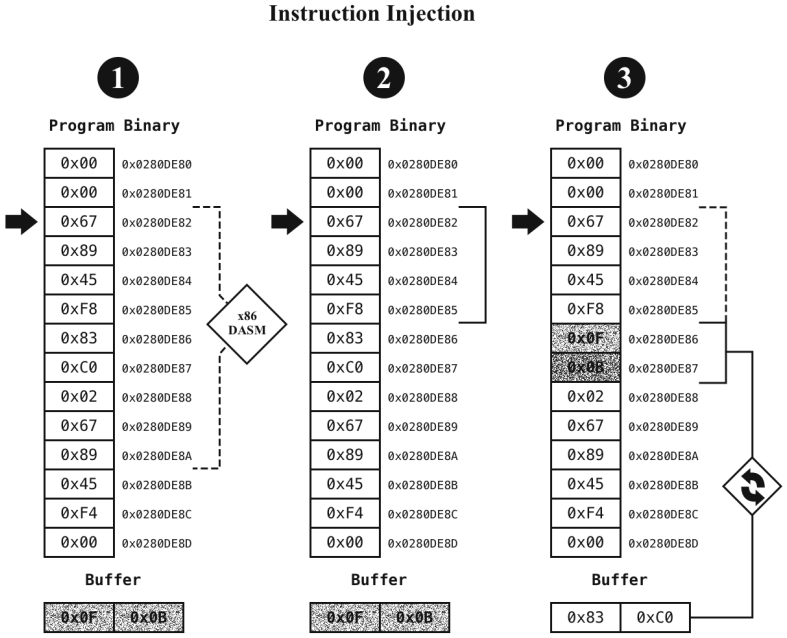

Figure 4.13: Injection of the UD2 illegal instruction.

Instruction Injection

The injected fault is a two-byte instruction named `UD2`. The instruction is intentionally undefined, and is typically used for software testing.

However, simply replacing the next instruction with a fault is not so trivial. Intel's x86-64 instruction set architecture (ISA) does not have a fixed length for each instruction. This meant that we cannot not simply write the faulty instruction to a location n bytes away. The instructions, of course, are also compiled machine code. It has no formatting or syntax to parse. The solution to this problem is to disassemble the instruction in the first handler, where the start of the faulting instruction is known. This technique was used with the Intel XED dis-assembler, which could provide both the opcodes and length of the machine code it disassembled. The length lets us compute the start of the next instruction, and replacement is now trivial. This is illustrated in Figure 4.13, where the dis-assembler in (1) is given a buffer of the largest possible size an x86-64 instruction can be. It then returns the true size of the instruction at the starting address, which is four bytes (2). This knowledge is used to inject the fault in (3). Note that the maximum length of the instruction is depicted to be 10 bytes for illustration purposes only; the actual length is 15 bytes.

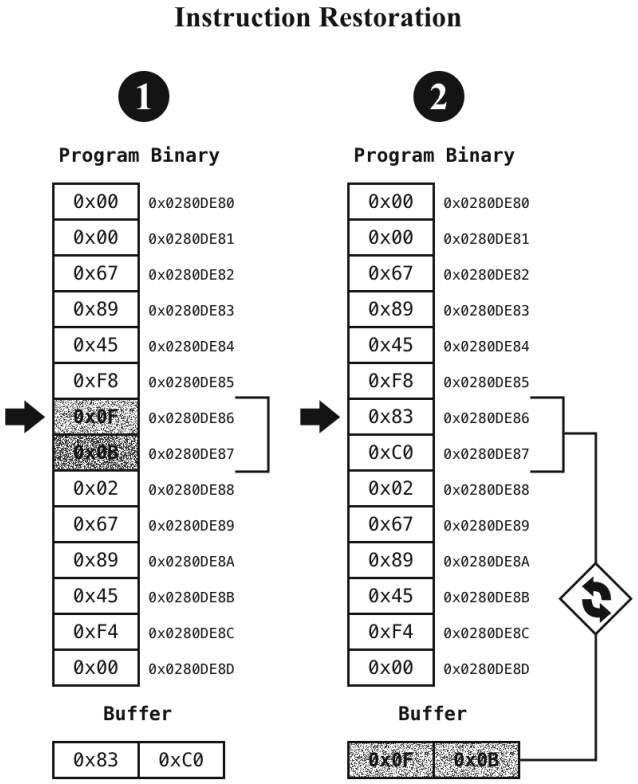

Figure 4.14: Restoration of original program code.

The injected instruction (`UD2`) raises a `SIGILL` signal. Naturally, the restorative logic was placed in a handler assigned to capture `SIGILL`. When the `SIGILL` handler is invoked, the replacement of the instruction is straightforward. Figure 4.14 illustrates the instruction restoration. First, the handler provides the address of the faulting instruction we want (1). Next, the original data is then swapped with it (2), and read-only protections reapplied. When the handler exits, it restarts the last instruction, and the program continues as normal.

At this point, it may have crossed the reader's mind that commandeering the signal handlers leaves the system open to a terrifying adventure were a genuine segmentation fault or illegal instruction to occur. Luckily, some thought has been put into this very problem, and the current mitigation strategies are in place:

1. **SIGSEGV**: When a segmentation fault is raised, the signal handler first checks whether or not the address of the fault lies within the range of the shared memory map that is being used by the DSM. If it does not, the program exits and informs the user that a segmentation fault occurred.

2. **SIGILL**: When a illegal instruction is raised, the signal handler checks if a flag has been set indicating that the program expects the illegal instruc-

tion. This flag is set after the fault was injected in the `SIGSEGV` handler. If the flag is not set, the program has performed an actual illegal instruction and exits, informing the user of the error.

Although capturing write accesses was by far the single most complex solution to a problem in the DSM, there remain countless other problems that took extensive amounts of time and thought to fix. However, to both remain punctual, and respect space constraints, these subjects will not be discussed here.

Evaluation

A series of example programs were used to evaluate the DSM. Some of these programs simply compared the effectiveness of different problem solving approaches in the DSM, while others pitted it against message passing library MPI to measure relative performance. The programs to be discussed include:

1. **Pingpong**: A "game" played between two processes where each process outputs "Ping" or "Pong" in lockstep. This program comes in two variants: a busy waiting version, and a semaphore based version.

2. **Primes**: Primes involves splitting the computation of the number of primes across a range, and distributing the work among an arbitrary number of processes.

3. **Contrast**: Contrast involves boosting the contrast of an image by splitting the task of computing the scaling factor and applying it across an arbitrary number of processes.

5.0.1 Pingpong

Pingpong was the first program tested with the DSM, and is primarily used to compare the efficiency of the messaging protocol's access operation against the use of semaphores. Two variants of pingpong were produced. The first, named "BusyWait", has each process constantly check a variable in shared memory to determine when to print. This variable is updated by one of the processes each time, allowing each to take turns printing their "Ping" or "Pong". The second variant, named "Semaphore", uses the semaphores provided by the DSM to mimic the lockstep behaviour.

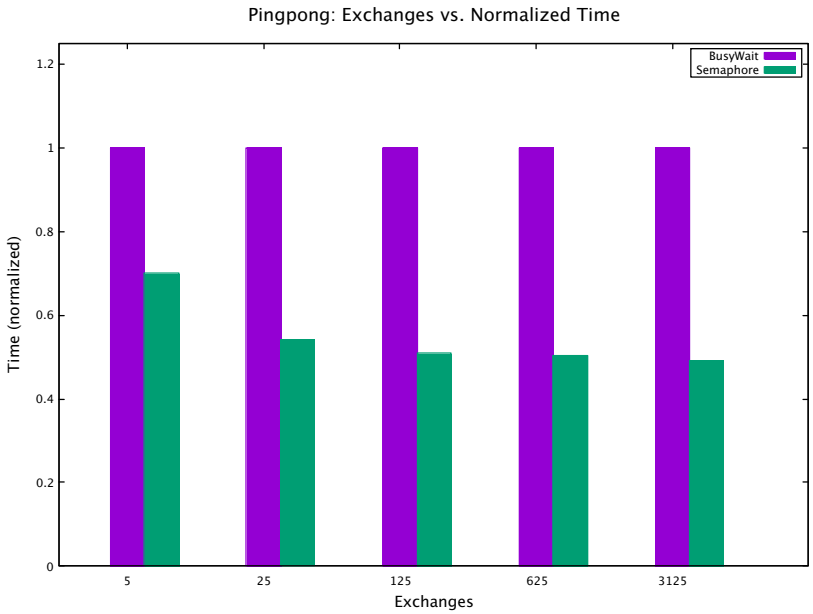

Figure 5.15: Comparison between busy waiting and semaphore based Pingpong. Data points averaged over five trials.

In Figure 5.15, the term "Exchange" refers to how many times a single process printed a "Ping" or "Pong". The total number of printed "exchanges" is thus two times that of the exchange. The figure shows how there exists a constant difference in the amount of time it takes to run the program for exponentially larger exchanges. This is explained by the fact that fewer messages are needed when manipulating semaphores as opposed to enforcing global accesses. It gives more weight to the observation that explicit synchronization tends to lead to better performance, at the cost of abstraction. In this case, the programmer may achieve a constant speedup, but only if the problem is "thought of" in terms of semaphores.

5.0.2 Primes

Primes is a problem that is simple to parallelize and scales well. This makes it a good choice for comparing different parallel computing solutions. In Primes, you are expected to compute the number of prime numbers across a range. This range is bound at zero on the lower end, and takes a user defined upper bound as input. To compute primes, each process calculates its own starting integer. This integer is computed as $2 \times i + 1$, where $0 \leq i < N$ is the process's global identifier from a collection of N worker processes. Each process computes the next number to test with $i_{j+1} = i_j + 2 \times N$. This method of distributing work makes it very easy to add or remove processes with minimal communication overhead. The reason for this is that the only synchronization required comes at the end of the computation when each process must contribute its local sum to a total.

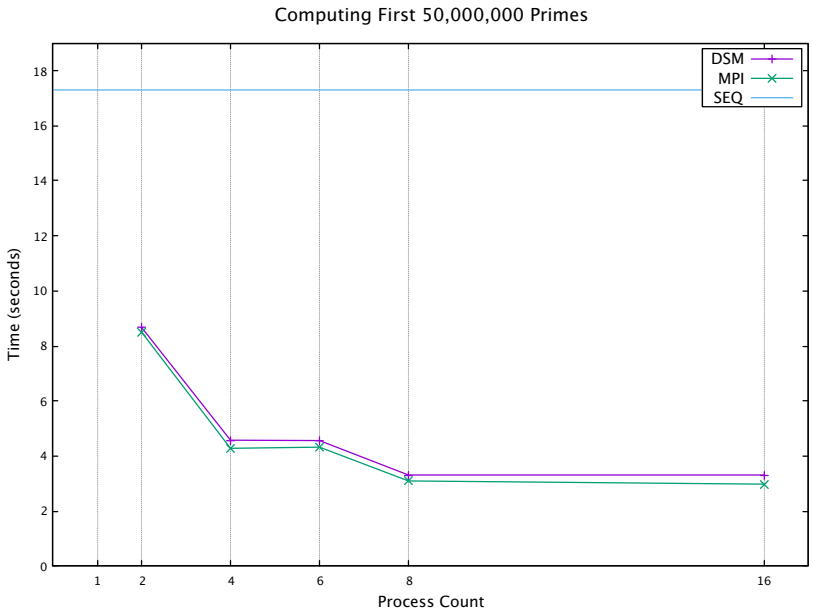

Figure 5.16: Comparison between MPI and DSM. Data points averaged over five trials.

Figure 5.16 compares how long a set number of processes took to compute fifty-million primes for the DSM and a message passing library called MPI. [15] Unsurprisingly, the DSM is slightly slower than MPI, with a more pronounced difference at higher process counts. This is also reflected in the efficiency comparison,

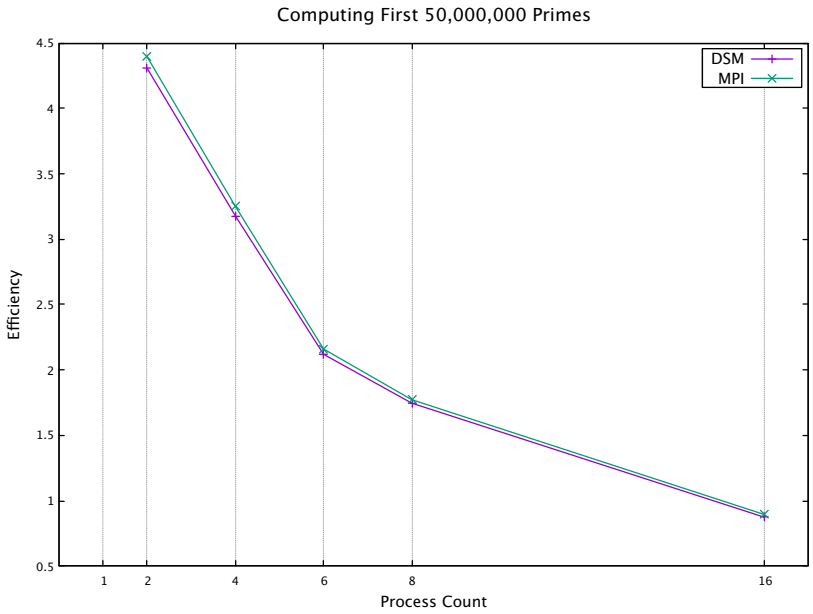

Figure 5.17: Comparison between MPI and DSM. Data points averaged over five trials.

(Figure 5.17) where the DSM appears to be less efficient than MPI by a small constant difference. MPI's triumph is likely explained in part by the efficiency of its messaging protocol. The divergence at higher process counts gives some weight to this hypothesis, given that more processes require more coordination, and thus more messages. As mentioned earlier, this difference is not surprising. MPI is a more mature platform, and fewer messages are needed when synchronizing the results compared to the DSM. It should be noted, though, that the development machine was only in possession of six cores, so results after the process count of six will not reflect that of a fully parallelized machine.

5.0.3 Contrast

Where the Primes example required only a single synchronization point, Contrast is a different story. The Contrast program does the following:

1. Exactly *one* process opens an image file, and loads the data into shared memory.

2. All processes compute a local minimum and maximum for their data segments.

3. The processes synchronize. At this point, a global minimum (min) and maximum (max) are found.

4. The processes all compute the same scaling factor as $scale = \frac{high-low}{max-min}$, where $high$ and low are the range over which the image contrast should

be stretched. *high* is typically given a value of 255, and *low* a value of zero.

5. The processes all boost the contrast of their respective data segments. Each pixel $im[i][j]$ is adjusted by first computing its baseline value, and then multiplying that by the scaling factor ($im[i][j] = (im[i][j] - min) \times scale$).

6. The processes synchronize once more.

7. Exactly one process writes the file to output.

Because the entire image is loaded into shared memory, the DSM ends up being far slower than even sequential variants of the program. In fact, a large amount of the test images reserved for this problem could not even be evaluated in a timely fashion due to the overhead generated by the DSM. The source of the overhead lies in the synchronization mechanism. Because each modified byte in the shared memory space is considered an access, five messages are sent for every change made. In a image tens of megabytes large, the DSM is rendered unusable.

5.0.4 The Problem of Large Data

The problem of synchronizing large amounts of data was one recognized early on in the development of the DSM. The issue stems from a lack of available context. In message passing, the programmer may choose to explicitly send large amounts of data around, resulting in few unnecessary messages. The DSM does not enjoy this luxury. Instead, it captures changes on a byte-by-byte basis. Each access requires the system perform a global synchronization; even when there is no need to do so. To solve the problem of bulk data transfers, three possible approaches exist:

1. **Additional Directives**: New functions may be added to the DSM's interface in order to facilitate the synchronization of large amounts of data. This, however, means the DSM begins to resemble message passing, where directives are used for all data transfer needs.

2. **Weaker Coherence Semantics**: If the consistency model of the DSM is weakened, it would allow for models where data may be written to memory without requiring global synchronization immediately. This means that when synchronization or global consistency was required, all changes could be synchronized "in bulk". This is more in the spirit of a DSM.

3. **Adjustable Shared Memory Spaces**: This solution is a hybrid between additional directives and weaker coherence semantics. In adjustable shared memory spaces, directives allow sharing to be explicitly turned on or off for segments of memory in the shared address space. This allows large changes to be synchronized across processes with far fewer messages. Data can be written to segments in bulk, and then synchronized in one go.

Ultimately, the performance of the DSM means it is only justified in situations where a small number of variables must be shared, and synchronizations are infrequent. The large synchronization overhead is a show-stopper for any

problem that requires manipulating large amounts of data within the shared memory space.

Conclusion

Distributed shared memory is one of two main paradigms for parallel computing on distributed systems. This paper has strategically reconstructed the process and design choices made for the implementation of a DSM as a software library. The consistency model, system architecture, and design challenges have been carefully documented in order to provide – as best as possible – a clear and concise template for future work.

6.0.5 Future Work

Given the complexity and breadth of design choices available for a DSM, there exist many ways in which to improve the current project. Some suggestions of these are made for the reader, though the list is by no means exhaustive.

- **Data Overhead Reduction**: As touched upon in Section 5.0.4, reducing the overhead of data movement to the shared memory space is a high value improvement. This may be done by introducing special directives to orchestrate large data synchronizations, adjusting the coherence semantics, or implementing a hybrid of the two.
- **Page Ownership**: Page ownership schemes are employed by many legacy DSMs, and involve splitting up the shared memory space into fixed length segments which can be owned by different processes. Any process which owns a memory segment may write to it uninhibited. However, if a process wishes to write to a segment owned by another process, it must first request it before becoming the owner. Page ownership allows a DSM to avoid unnecessary synchronizations between processes performing unrelated work. [4]
- **Data Compression**: Data compression, such as run-length encoding, allows certain forms of data to be compressed losslessly before transmission. Although only useful in data with many sequences of similar elements, it could potentially reduce the overhead of synchronizing large segments of data. [14]
- **Message Queues**: The current DSM implementation does not have any message queuing. Both the arbiter and server rely on fixed-size socket buffers provided by the kernel to perform this task. The lack of message queues leaves them vulnerable to failing if hit with a large influx of messages. Implementing message queuing would allow the system to continue operation in situations where the socket buffers would be overwhelmed.

- **Fair Semaphores**: The current implementation of the DSM does not unblock semaphores fairly. In a system with fair semaphores, a post to a semaphore on which more than one process is blocked should unblock the process which has been waiting the longest. In short, the blocked processes should form a queue, and the head of the queue is unblocked and dequeued each time a post occurs. In the DSM, processes are not queued, but marked as blocked on a semaphore within the process table. The process chosen to be unblocked is the first one found in the table. This makes the system biased towards processes of arbiters who connected earliest, and towards processes of that arbiter which connected latest.

In its final state, the project totals 2341 source lines of C, which is split among 16 source files. The project also includes several example programs and testing facilities, most of which were used in both the evaluation and development of the program. Finally, extra effort has been put into ensuring that the project source code is clean and well commented, so that future changes can be made with as little pain as possible.

Source Code

The project source code is available at https://github.com/Micrified/dsm

Bibliography

[1] T. S. Alexrod, "Effects of synchronization barriers on multiprocessor performance", *Parallel Computing 3*, pp. 129–129, 1985.

[2] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends", *Domesticating Parallelism*, no. -, pp. 26–28, 1986.

[3] "Operating system enhancements for distributed shared memory", *Advances in Computers*, vol. 39, pp. 197–200, 1994. DOI: https://doi.org/10.1016/S0065-2458(08)60380-0.

[4] M. R. Eskicioglu and T. A. Marsland, "Distributed shared memory: A review", *Department of Computer Science: University of Alberta*, no. Technical Report TR 96-22, pp. 1–10, 1996.

[5] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation*, 2nd ed. Prentice-Hall, Inc, 1997, pp. 66–68, ISBN: 0136386776.

[6] V. G. Oklobdzija, *The Computer Engineering Handbook*. CRC Press, LLC, 2001.

[7] R. Love, *Linux System Programming*. O'Reilly Media, Inc, 2007, ISBN: 0596009585.

[8] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems Concepts and Design*. Pearson Education, Inc, 2011, pp. 106–107, ISBN: 0132143011.

[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual consistency", vol. 12, no. -, pp. 3–7, 2014.

[10] N. Ishkov, *A complete guide to linux process scheduling*, 2015.

[11] M. Kerrisk, *Posix shared memory*, man7.org, 2015.

[12] B. Hall, *Beej's guide to network programming*, 2016, pp. 39–51.

[13] *Mpi_comm_rank(3)*, 2017. [Online]. Available: https://www.open-mpi.org/doc/v2.0/man3/MPI_Comm_rank.3.php.

[14] (2017). Run-length encoding, Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Run-length_encoding.

[15] (2018). Mpi, The Open MPI Project, [Online]. Available: https://www.open-mpi.org.