Three Flavors of Paxos

Kat Cannon-MacMartin

April 23, 2021

Contents

1 Introduction

Distributed computing is a vast field, encompassing a wide range of problems, solutions, and modeling tools to bridge the two. While specific distributed systems are each subject to specific models that define their particular behavior, generalized models of these systems can usually be grouped into three broad categories. These categories, often called "viewpoints," are parallel algorithms with shared memory, parallel algorithms with message-passing, and distributed algorithms with message-passing.

In an effort to illustrate both the differences and similarities between these models, the code example accompanying this paper implements the *same* algorithm, designed to solve the *same* problem across all three viewpoints. Though a number of algorithms would suit this task, Leslie Lamport's classic Paxos algorithm, first described in [1], is a natural choice. In this case, each implementation of Paxos solves the traditional problem of leader election, in which the processes on a system must agree on a single process from which to take orders.

2 Theoretical overview

2.1 The viewpoints explained

The three implementations of Paxos included here are each designed to address a specific "viewpoint," or model for how a distributed system works. To get an idea of how these cases differ, each viewpoint is defined as follows.

2.1.1 Parallel algorithms with shared memory

In this model, processes communicate using a section of shared memory. Each process is assigned a block to which it is allowed to write, and it writes only to its own block. All processes can read all blocks, however, allowing for communication.

Processes also run in "parallel", meaning they function as organized, co-operating actors in a unified system. Explicitly, this means that each process has knowledge of the network topology and status of other processes. For clarity, this can be contrasted with the "distributed" viewpoint (explained in full below), in which processes have no knowledge of network topology, and rely entirely on the communication they receive to determine the status of the system.

2.1.2 Parallel algorithms with message-passing

Like the previous model, processes run in parallel. In the case of message-passing however, knowledge of network topology plays a much more significant role. Instead of utilizing a shared block of memory for communication, processes in this model send each other messages, using process ID numbers as addresses.

Because of the parallel nature of the system, processes can be arranged into a specific network topology ahead of time, and consequently all processes have knowledge of this structure. In practice, this means that process a is aware of the existence of process b as soon as it begins execution, and can therefore immediately address and send messages specifically to process b.

2.1.3 Distributed algorithms with message-passing

Just like the previous model, processes in this model communicate by passing messages. Unlike a parallel system, however, processes in a distributed system have no knowledge of the network topology or specific addresses of other processes. When process a begins execution, it knows only that N other processes exist on the system. Process a is not aware of the existence of process b, and consequently cannot address messages specifically to process b until it receives a message from b.

With this added layer of complication, processes in a distributed system have two options for sending messages. Though they it may not be aware of every process in the system, process a can flood the system with messages, sending $N - 1$ messages, one to each other process. Process a can also send specifically addresses messages to any process b from which a has first *received* a messaged.

2.2 A summary of Paxos

The Paxos algorithm designed by Leslie Lamport is a generalized algorithm for reaching agreement on a value between processes. At the core of the algorithm are three roles: the proposer, the acceptor, and the learner. A simple execution of the algorithm proceeds as follows.

First, the proposer sends a *prepare* message to all acceptors. This message includes a unique identifying number n, called the ballot number, but no value just yet. In return, each acceptor replies to the proposer with a *promise* message using the same ballot number n. Additionally, if the acceptor has previously accepted any value v, the promise message

will include that value along with its corresponding ballot number. By issuing a promise, an acceptor promises that it will only accept messages with ballot number n or greater. Consequently, if an acceptor has already issued a promise for a ballot number greater than n, it simply ignores the prepare message.

If the proposer receives a promise from a majority of acceptors, it may then issue a *propose* message, imploring the acceptors to accept a proposed value. In order to decide on the value to be proposed, the proposer reads all received promise messages. If any acceptor included a previously accepted ballot number and value, the proposer chooses the greatest of these ballot numbers, and must propose the corresponding value. In the event that no ballot number, value pairs are returned, the proposer may choose its own value to propose. The proposer then issues the propose message to all acceptors.

Upon receiving a proposal, an acceptor checks if the proposal's ballot number is equal to or greater than its promised ballot number. If it is, the acceptor accepts the value in the proposal, updating its internal record of accepted ballot number and value. It then sends an *accept* message to the proposer, informing it that the proposal has been accepted. The acceptor also sends *teach* messages to any learner processes, informing them of the newly accepted value.

Learner processes serve as verification of acceptances. Once a learner is informed of the same value by enough acceptors via *teach* messages, it updates the status of the system in accordance with the decided upon value.

2.3 Paxos for leader election

The traditional description of Paxos relies on the defined roles of proposer, acceptor, and learner. In a leader election problem, however, the exact issue is the lack of defined roles. Processes may have knowledge of the topology of the network, or identifiers of other processes (depending on the viewpoint), but before election occurs, no single process will be considered the leader.

In this case, as is often the case in Paxos implementations, every process will play all three roles. The value to be agreed upon is the ID of a single process to be considered the leader. Each process begins execution by attempting to institute itself as leader, before gradually deferring to the single process that is eventually chosen.

In each of the accompanying implementations, the general procedure for an individual process proceeds as follows.

At initialization, a process p sends a prepare message with ballot number n. While

awaiting promises in response to that prepare, if p receives a prepare statement with a ballot number greater than n, it abandons its own prepare and instead takes the roll of acceptor, returning a promise in response. Process p will then ignore any promise messages it receives corresponding to ballot number n.

In the even that p is *not* interrupted by a higher-number prepare, it awaits a sufficient number of promise statements. Upon receiving these promises, p issues a proposal, proposing either itself as leader, or one of the leader ID values received in a promise. While awaiting accept statements in response to this proposal, p can once again be interrupted by a higher-numbered prepare statement, in which case it aborts the current proposal and ignores all corresponding accepts.

Any process p_i that accepts the proposal of p issues two messages in response. One accept message to p, informing it that its proposal has been accepted, and one teach message. In this case, there is no specified learner role in the algorithm. Instead, p_i sends a special teach message, called $LEADER$, to the process ID that it has accepted as the leader.

Every process in the system keeps a running tally of the number of $LEADER$ messages it has received. If a process receives $LEADER$ from a majority of processes in the system, it is recognized as the leader.

2.4 Leader election Paxos in each viewpoint

While the above description lays out the generalized Paxos algorithm, as well as the specific form used for electing leaders, it is still an abstraction removed from the specific details of implementation. While "messages" are mentioned, they are merely description of communication between processes, and need not correspond to literal messages. Furthermore, according to the topology of a specific network of processes, communication need not always occur in the exact way described above. Below are specific descriptions of how leader election Paxos behaves in each of the three viewpoints addressed by the accompanying code example.

2.4.1 Distributed with message passing

This viewpoint is addressed first, as its specific implementation is most similar to the general description of Paxos supplied above. Messages are used for communication, and they are distributed in exactly the way described by the generalized algorithm. Because of the distributed nature of the system, messages sent a proposer are not addressed to specific processes, but instead flooded to every other process on the system. The first of these mes-

sages is obviously the prepare message, and it is via receiving this message that all acceptors learn the identity and address of the proposer. Discovery of this address allows the specific addressing and return of promise messages where applicable. Likewise, receiving promises allows the proposer to discover the addresses of acceptors. Note, however, that a process in the proposer role need never send individually addressed messages anyway.

True to the generalized Paxos description, all prepare and propose messages are broadcast to every process on the system, while promise and accept messages are addressed individually to the relevant proposer. Teach, aka $LEADER$, messages are addressed to the accepted leader because the value being accepted is itself the individual address of the leader process.

In this implementation, upon receiving a majority of $LEADER$ messages, the agreed upon leader issues an END message to all processes in the system. If the algorithm has executed correctly, all processes will respect the process issuing END as leader and obey they command, ending their execution.

In a system with N processes, this implementation requires $3N + \frac{N}{2} - 2$ messages for a complete, unaborted round. Prepare and proposal statements each require $N - 1$ messages. For a majority to be reached, $N/2$ messages are required for both promises and accepts. Finally, $N/2$ $LEADER$ messages are required. This gives

$$2(N-1) + 3(\frac{N}{2}) = 2N - 2 + N + \frac{N}{2} = 3N + \frac{N}{2} - 2$$

which is a linear $\mathcal{O}(n)$ message complexity for a single round.

2.4.2 Parallel with message passing

Parallel processes, and consequently the option of a structured network, provides a plethora of options for implementing Paxos. The implementation described here is by no means the definitive choice, or even a common one. In this case the network of processes is pre-structured as a ring, with each process passing messages only to its neighbor in a clockwise direction. While a ring structure is far from the most efficient network structure, and by definition not tolerant to any faults, it is a simple and theoretically pure model for how a structured network can affect the flow of an algorithm.

The implementation described below, though mostly original, is in part influenced by an algorithm for Paxos in a ring described in [4]. The protocol for processing messages is also informed by the algorithms for leader election in a ring found in [5].

While the distributed form of message-passing Paxos reaches majority consensus via mass

prepare/propose statements with replies, the parallel message algorithm aims to achieve total consensus by relaying prepare/propose around the ring. Processes are arranged in the ring by ascending ID number, so process p_i has clockwise neighbor p_{i+1} and counter-clockwise neighbor p_{i-1}. The algorithm proceeds as follows.

A proposer process p_i issues a prepare statement with ballot number n to its clockwise neighbor p_{i+1} in the ring of processes. If p_{i+1} has not already promised a higher ballot number than n, it issues a promise message to its own clockwise neighbor, p_{i+2}, *not* back to the proposer p_i. If p_{i+1} has previously accepted a ballot number and value pair, it includes that pair in the promise message. When p_{i+2} receives the promise from p_{i+1}, it treats the message as if it were a prepare statement. If p_{i+2} has promised a higher ballot number, it ignores the message, otherwise it forwards the promise to the next process in the ring, p_{i+3}. Before forwarding the promise, p_{i+2} checks the previously accepted ballot number, value pair in the message against its own. If its own previously accepted ballot number is greater, it replaces the pair received from p_{i+1} with its own. If p_{i+2}'s previously accepted number is less, or it has none at all, it simply forwards the values received from p_{i+1}.

In this way, the prepare statement sent by p_i proceeds around the ring, stopping if at any point a process has already accepted a higher ballot number. If p_i receives a single promise in return, from process p_{i-1}, it means the prepare has traveled all the way around the circle without fault. Furthermore, the promise statement p_i receives from p_{i-1} includes only the previously accepted ballot number, value pair with the highest ballot number. If this pair is present, p_i will propose the value from it. If it is not, p_i will propose itself as leader.

Just like the prepare and promise statements, p_i sends its proposal only to p_{i+1}. If p_{i+1} accepts, it forwards the accept around the ring. In this case, there is no need for a teach/$LEADER$ message to be sent by every acceptor. If the forwarded accept message is returned to p_i, it knows its value has been accepted. If p_i proposed itself as leader, the algorithm is complete. If it proposed a different process as leader, it forwards a single $LEADER$ message around the ring to the process chosen.

In a system with N processes, this implementation requires on average $2N + \frac{N}{2}$ messages for a complete round. Prepare and proposal statements require N messages each to traverse the ring, while an average of $\frac{N}{2}$ $LEADER$ messages are required to forward the result in the event that a proposer does not choose itself as leader. Like the previous implementation, the complexity is a linear $\mathcal{O}(n)$. However, while distributed Paxos requires approximately N more messages to reach majority consensus, parallel Paxos in a ring reaches total consensus using less messages.

2.4.3 Parallel with shared memory

While Paxos in a ring deviates from the generalized description in the interest of utilizing a network structure, shared memory Paxos deviates significantly further. Unlike either message-based implementation, processes in shared memory Paxos are not required to wait for the next message to proceed in their execution. Instead, processes in this implementation rely on constant reads of a shared memory block to simulate "communication" with other processes. This form comes with some pitfalls, however.

According to a significant result of Fischer, Lynch, and Patterson in [6], a completely asynchronous, non-blocking consensus algorithm is not possible. In each of the previous implementations, processes block while they wait for messages. In the shared memory model, however, a process can read the shared memory segment at any time, as many times as it wishes. Consequently, a single process could theoretically complete all of its reads and decide that consensus has been reached before any other process completes even one read.

The following implementation of shared memory Paxos is a modification of the Disk Paxos algorithm proposed by Eli Gafni and Leslie Lamport in [3]. In that paper, the authors recommend using real time clocks during leader election, to simulate blocking of processes and ensure a correct execution. Consequently, real time clocks have been utilized in this implementation of shared memory Paxos, as described below.

Without the explicit communication method of message passing, shared memory Paxos instead relies on each process maintaining a block of information in the segment of shared memory. In a system with N processes, the shared memory block consists of N blocks b_0 through b_{N-1}. A process p_i may read any block, but may only write to block b_i.

Each block b_i consists of three values pertaining to its respective process p_i. First, the current ballot number n of p_i. Additionally, if p_i has previously accepted a value, the block b_i contains the pair n', v' of the previously accepted value v' and its corresponding ballot number n'.

At the start of a round, a process p chooses a new unique ballot number n, just as a proposer does in previous implementations. Process p then proceeds to read the entire block of shared memory, comparing its own ballot number n against the current ballot number n_i of each block b_i. In the event that p encounters a ballot number higher than its own, it aborts and sleeps for a random number of seconds. This is the real-time clock component of the algorithm. When p wakes from the sleep, it will start a new round with a higher ballot number.

If p does not abort, it returns from the read with a record of every pair of previously

accepted values and ballot numbers n', v' from all blocks that it read. If it encountered no such pairs, p chooses its own ID as the value v to "propose." Otherwise, p chooses the value v' with the highest corresponding ballot number n' to serve as v, as in the other implementations of the algorithm. At this point, p updates its own pair n', v' to the current values of n, v.

In the second, "propose," phase of the algorithm, p once again reads all blocks, again comparing all ballot numbers to the same ballot number n. In the event p encounters a greater ballot number, it compares its proposed value v with the previously accepted value v' of that block. If $v = v'$, p proceeds with its read. If $v \neq v'$, p aborts and sleeps for a random number of seconds.

If process p completes this second round of reads, it has decided on a value. Instead of communicating this decision to other processes, p simply exits.

Unlike the message-based implementations, processes in shared memory Paxos operate almost entirely independently, simply reading the states of other processes to inform their own decisions. It is the protocol for "inheriting" the previously accepted ballot number and value pairs n', v' that ensures all processes arrive at the same value.

The complexity of shared memory Paxos cannot be easily compared to message-based alternatives, as memory read/writes are not the same actions as message passing. However, one completed "round" of shared memory Paxos for all processes requires $2(N-1)$ reads per process, and exactly one write per process for a total of

$$N \cdot 2(N-1) = 2N^2 - 2N$$

reads and N writes. This gives a read complexity of $\mathcal{O}(n^2)$ and a write complexity of $\mathcal{O}(n)$.

3 Implementation

The code example accompanying this paper demonstrates leader election Paxos according to the implementations described for each of the three viewpoints. The entire project is written in C, and a unique build system is employed to mix-and-match source files when creating each executable. The provided source code compiles into three executables, *bin/message_paxos*, *bin/ring_paxos*, and *bin/memory_paxos*. Each executable corresponds to one of the implementations described above.

3.1 General implementation details

All three executables employ the POSIX pthreads module to spawn individual processes. The two message-passing implementations use POSIX messages to communicate, while the shared memory implementation simply employs a common block of allocated memory that is passed to every thread at initialization.

While running, each thread p prints status updates when any of the following events occur:

- p sends a prepare or propose message, or starts a new round of reading in memory Paxos.
- p receives an accept, promise, or $LEADER$ message, or aborts a read in memory Paxos.
- p recognizes itself as leader, or receives an END message, or recognizes another process as leader in memory Paxos.

Finally, upon exiting, each process prints the process it recognizes as leader.

By default, each executable runs with 5 processes. Passing a number n as an argument when calling the executable will instead attempt to run Paxos with n processes. However, the maximum number of processes is unfortunately quite low at 8. While the provided implementations allow for proper execution with many many more participants, this limit is imposed by the limit on POSIX message queue size. By default, POSIX message queues may not hold more than 10 messages for unprivileged processes. If a process attempts to send a message to a full queue, the message is lost. In fact, running a message-based implementation with only 8 processes already involves the loss of some messages, however the Paxos algorithm can tolerate these losses. Using 9 or more processes typically results in almost half the messages being lost, and under these conditions successful execution of the algorithm is at best extremely unreliable.

3.2 Build system details

The code example of these three flavors of Paxos attempts to express their unification through a unified build system. In fact, each implementation utilizes at most two completely unique source files, with all other sources shared between the three.

At the top level, one unified driver program, defined in *driver.c*, serves as the main function for all three executables. This function simply spawns threads which each execute the

node routine defined in *node.h*. Despite the unified header file, There are two different source files corresponding to *node.h*. *message_node.c* and *memory_node.c*, the former being used by both message passing implementations, the latter used by the shared memory implementation.

This is the point of great unification, however. Regardless of which **node** routine is used, both call the exact same **paxos** function defined in *paxos.c*. This bare-bones implementation of the Paxos algorithm control flow is shared by all three contrasting implementations. The algorithms differ in the subroutines called by **paxos**. Outlined in the header file *paxos_functions.h*, these functions such as **prepare**, **propose**, and **accept**, are implemented uniquely for each variant. This is the only point in the build process where all three implementations use a unique source file.

Besides the general utility functions, three baseline source files provide all the functionality required for the three algorithms. These are *threads.c*, used by all three implementations, *messages.c* used by both message passing implementation, and *memory.c* used by shared memory Paxos.

The *Makefile* found in the *bin/* sub-directory provides a clear breakdown of exactly which source files are used for each executable.

4 Conclusion

The same algorithm, solving the same problem of leader election, provides an excellent contrast between these three different viewpoints of distributed computing. At the highest level of theoretical abstraction, the three models are in many ways the same. Some number of processes must communicate to agree on a value, but the order in which they communicate and the speed at which they do so cannot be predicted. Every model requires an insurance that no process can get too far ahead of the others, as well as an insurance that the processes will not completely deadlock and halt progression.

These shared issues manifest differently in each case. In the case of message passing, the algorithm forces each process to wait for messages, providing an insurance that no process can complete unless a majority have at least proceeded in their execution. In the shared memory model, this insurance must be enforced via arbitrary waits.

These differences are still just specifics of implementation, however, and with enough abstraction a single algorithm, generalized Paxos, solves all three cases. This is the beauty of a well designed distributed algorithm. Regardless of how processes communicate, or how

they are structured in a system, the core protocol of increasing ballot numbers, coupled with inheriting previously accepted values, always finds consensus.

References

[1] L. Lamport, The Part-Time Parliament, *ACM Transactions on Computer Systems* **16**, 2 (1998), 133-169.

[2] L. Lamport, Paxos Made Simple, *lamport.azurewebsites.net* (2001), accessed 2021.

[3] E. Gafni, L. Lamport, Disk Paxos, *lamport.azurewebsites.net* (2002), accessed 2021.

[4] P. J. Marandi, M. Primi, N. Schiper, F. Pedone,
Ring Paxos: A High-Throughput Atomic Broadcast Protocol, *IEEE/IFIP DSN* (2010).

[5] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann (1996).

[6] M. J. Fischer, N. Lynch, M. S. Patterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM* **32**, 2 (1985), 374-382.