

ARPE final report

Samuel Arzac, supervised by Chung-Kil Hur
Seoul National University, Software Foundations Lab

October 2022 - July 2023

Context We are applying a new verification framework (CCR) implemented in Coq to a real-world network consensus program (Raft) to show its capabilities and improve on the previous safety proofs of implementations of this algorithm.

Problem We are verifying a program using conditional contextual refinement, which involves defining several layers of more abstract programs, the most abstract layer being proven correct. The specificity of this framework compared to usual contextual refinement is the addition of separation logic to restrict the setting in which programs are abstracted.

The challenge is to define these layers and prove that each one properly abstracts the previous one. This framework allows us to prove a Raft implementation in a more modular way than existing proofs.

Since it is a real-world program, there are also some technical challenges to be faced in Coq due to the size of the project.

Contributions We have partially written defined the first abstraction layers for the proof. Much of the work was focused around making tools to write and prove the abstractions. Those tools should make future progress and projects much easier.

We have defined a network model suited to the hypotheses of the algorithm, which can be used to prove a network of servers implementing Raft.

Perspectives We will continue proving the refinement between successive layers, to complete the refinement proof, and then prove the safety of the top layer.

Another objective would be to prove an implementation of Raft that is more widely used than the one we are proving.

Finally, we could use more advanced models of memory and concurrency in the proof.

Acknowledgements

I would like to thank Chung-Kil Hur for giving me the opportunity to work in his team, and for his supervision during my internship.

I would also like to thank the lab students for welcoming me and helping me settle down in Korea.

1 Introduction

Contextual refinement is method of proving that a given program is abstracted by another one, allowing to transfer properties from the abstract program to the more concrete one. This has a number of advantages, most notably that this abstraction can be done in several steps, using several “layers”, and that there is no restriction in terms of the supported features in the program.

Conditional contextual refinement (CCR)[10] is a verification framework combining the properties of refinement with separation logic. Though some work as already be done on this subject, it is the first framework to add separation logic while keeping the full flexibility of contextual refinement. The main target of this project is to demonstrate the power of this framework on a real-world program.

We are applying it to an implementation of a network consensus algorithm used for a network of servers to agree on a value. It allows clients to communicate with a network of servers at if it was only exchanging with a single server, but allows for some down time from the servers on the network. We started with Paxos[7], but moved on to Raft[8] since it is easier to understand and widely used.

Such verifications already exist, such as IronFleet[3] and WormSpace[9] for Paxos and Verdi[12] for Raft. We are proving a simple implementation of Raft written in C[11].

Our verification will have the benefit of being more modular and a more long-term objective is to have a verification with fine-grained concurrency, supporting the verification of some compiler optimisations.

The proof is still a work in progress, as applying the framework to a program of this size causes some technical problems to be solved in order to get to the theoretical aspects of the abstraction.

2 Preliminaries

2.1 Framework

2.1.1 Contextual refinement

Contextual refinement is a method of giving a specification to a program in the form of another program.

Definition 1. *Given two programs I and A , I contextually refines A if for any context, the behaviors of I in this context are included in those of A in that same context.*

It is noted $I \leq_{ctx} A$.

We call I the “concrete” program and A the “abstract” program.

Where the behaviors of a program are as follows.

Definition 2. *The behavior of a program is a set of traces, which are lists of observable events, ended by a return value, an error, a divergence without observable event, or a termination caused by the user (called Partial).*

The first advantage of contextual refinement is that all programming features are supported, since we use other programs for specification.

The second advantage is compositionality:

Theorem 1. *Contextual refinement supports horizontal and vertical compositionality:*

- *horizontal compositionality:*

$$I_1 \leq_{ctx} A_1 \wedge I_2 \leq_{ctx} A_2 \Rightarrow I_1 \circ I_2 \leq_{ctx} A_1 \circ A_2$$

where $\circ$ is the program composition.

- *vertical compositionality:*

$$I \leq_{ctx} M \wedge M \leq_{ctx} A \Rightarrow I \leq_{ctx} A$$

However, a drawback of contextual refinement is that it isn't possible to have restrictions on contexts, all contexts must be considered.

2.1.2 Conditional contextual refinement

Conditional contextual refinement[10] is a framework adding the option to add pre-conditions and post-conditions with separation logic to the programs while keeping both compositionality and programming features.

The main idea is to encode conditions in the code of the program, and then prove the contextual refinement.

Given a separation logic specification S for a program A , the specification is embedded into the program to create a wrapper noted

$$\langle S \vdash A \rangle$$

Definition 3. I refines A under the specification S if $I \leq_{ctx} \langle S \vdash A \rangle$.

Since the same notion of contextual refinement is used, the compositionality properties still hold.

This encoding works by *assuming* the pre-condition and *guaranteeing* the post-condition. If an assumed (guaranteed) proposition is false, then it triggers *undefined behavior* (*no behavior*). Conversely, when a function is called in the program, the wrapper guarantees the function's pre-condition before the call and assumes its post-condition after.

Definition 4. If a program has an undefined behavior (*UB*), its behavior is the set of all possible traces.

Definition 5. If a program has no behavior (*NB*), its behavior is the singleton $\{[Partial]\}$.

This means that when a context falsifies an assumption in the abstract program, its set of behaviors is the set of all possible behaviors, thus the refinement trivially holds. In the case of a failed guarantee in the abstract program, its behavior is the empty set, so the refinement only holds if the concrete program also has no behavior. The idea is that assumptions restrict the contexts where actual refinement proofs will be made, and guarantees are used to call other programs and prove that their assumptions are satisfied and to give a post-condition for other programs to rely on.

Conversely, in the concrete program a failed assumption falsifies the refinement unless the abstract program has an undefined behavior, and a failed guarantee makes the refinement trivial.

When composing all necessary programs together, we can cancel assumptions following guarantees with similar proposition, and obtain an abstraction without wrappers.

This framework is implemented in Coq. It allows the definition of programs using interaction trees[13], and give them a separation logic specification using Iris[6].

2.2 The Raft algorithm

This section is an explanation of the Raft algorithm, from [8].

2.2.1 Overview

Raft is a consensus algorithm for a network of servers that need to agree and choose a sequence of commands to be applied to a state machine. The idea is that a client doing a request to the servers will see no difference with communication with a single server, but some servers failing in the network will not cause the service to be unavailable.

More precisely, we have the following assumptions on the servers:

- They may stop at any moment.
- They may restart.
- They compute at arbitrary speeds.

We also have assumptions on the network:

- Messages can be lost.
- Messages can be reordered, duplicated and take an arbitrary time to arrive.
- Messages cannot be corrupted.

In these conditions, we want the servers to agree on a sequence of commands, known as the **log** requested by clients. These commands have to be applied to the state machine, to then send an answer to the client.

The guarantees are the following:

- The algorithm is **safe**: it never returns an incorrect result to a client.
- The service is available as long as a majority of servers are running and can communicate.

2.2.2 Roles

During the execution of the algorithm, each server has one of three roles: **follower**, **leader** or **candidate**.

- The **leader** is the server receiving the client requests. It adds these requests to its log then tells other servers to do the same.
- **Follower** is the default role, a server with this role replicates the log of its leader.
- When a follower has no information about a leader (due to a communication issue or a crash from the leader), it becomes a **candidate** and asks other servers for their votes.

The algorithm execution is divided in several phases, known as **terms**. When a follower does not receive any message from a leader for a given amount of time, it starts an election by becoming a candidate with a new term number. Any server receiving a message with a higher term number than its own updates its term number.

2.2.3 Server data

Each server holds the following data in stable memory. It should be recovered if a server restarts after crashing.

- **currentTerm**, its current term number.
- **votedFor**, the server it voted for in the current term (if any)
- **log**, its log, which is a sequence of commands. Commands themselves are not relevant to the Raft algorithm, however each is associated with a unique index and the term number it was added at.

In addition, they also store the following data in volatile memory.

- **commitIndex**, the last log entry index that has been *committed*, meaning that a majority of servers have it in their log, so it can be applied to the state machine safely.
- **lastApplied**, the last log entry index applied to the state machine. Servers will always try to have **lastApplied** catch up with **commitIndex** by applying commands to their state machine.

2.2.4 Leader election

Each follower has an election timeout value. If the time since the last message from a leader is above this value, it turns into a candidate, increases its term number, and triggers an election.

Triggering an election means that the candidate sends a message to all other servers asking for their vote. This message contains the candidate's term number, as well as the index and term number of its last log entry. When a server receives a vote request, it does the following:

- If the term number mentioned in the vote request is lower than its **currentTerm**, the server does not accept the request.
- Else, if the log of the candidate is at least as up to date as the receiver's, and if the receiver hasn't voted for another server for this term (known by looking at **votedFor**), then the receiver grants its vote to the candidate. A log is more up to date than another if the term number of its last entry is higher, or if it is longer in case of equal terms.

If a majority of servers grant their vote to the candidate, then this candidate becomes the leader for this term.

The election timeout is chosen with some randomness, to avoid all servers turning into candidates at the same time when the leader crashes, as this would result in them being stuck in a loop of election failures.

2.2.5 Appending entries

Clients can send commands to be applied to the state machine. When a leader receives one, it is added to its log. The server then has to replicate its log to all its followers.

Leaders regularly send **AppendEntries** messages to other servers to keep their followers updated on its log. Even when a follower is up-to-date, the leader will send a message as a heartbeat to prevent it from triggering an election.

For each follower, the leader keeps the following information:

- **nextIndex**, the index of the next entry needed to be sent to that follower. Initialized at the leader's last index + 1.
- **matchIndex**, the index until which the leader knows the follower is up-to-date. Initialized at 0.

If there is a N such that $N \leq \mathbf{matchIndex}$ for a majority of servers, if N is greater than the leader's **commitIndex**, and the term of the entry at index N is equal to the leader's **currentTerm** then the leader updates its **commitIndex** to N . This means that the current leader has proposed the entry and a majority of servers have accepted it, so its command can be applied to the state machine.

An **AppendEntries** message contains the entries that the follower needs to add to its log (none if the message is just a heartbeat), as well as the index and term of the entry after which the entries need to be added. The follower does the following when receiving an **AppendEntries**:

- If the leader's term is lower than the follower's, the follower rejects and informs the leader of its term. The leader will then know its term is outdated, and will turn into a follower with the new term number.
- If the index and term given for the previous entry do not match the log of the follower, the follower rejects. This will inform the leader that there is a mismatch between its log and the follower's, so the leader will decrease the **nextIndex** for this follower and try again.
- Otherwise, the follower adds then entries in the message after the given index. If the follower already has some entries after that index, all of them are deleted. In addition, the message contains the leader's

commitIndex. If the follower's **commitIndex** is lower than the leader's, it is increased to $\min(\text{leader's } \mathbf{commitIndex}, \text{follower's last log index})$.

2.2.6 Other functionalities

Raft supports changes in configuration, such as adding or removing servers. It also supports snapshots, which consists in compacting the committed entries in the log to save some space. For simplicity we will ignore those functionalities, since we have chosen to remove them.

2.3 ADO

Atomic Distributed Object (ADO)[5] is a model for distributed systems expressing the differences between the states of the different servers. It can be used to refine some protocols such as Raft or multi-Paxos. We will use this model as our top abstraction in the refinement.

This model keeps a global state of the network using a tree with the history of all events. If the tree has multiple branches, it means that some servers do not agree on the history of events.

The operations a server can do are detailed in the following sections.

2.3.1 Election

In ADO, a server can call *pull* to acquire the leader role. The outcome of this election is non-deterministically decided by an oracle.

The oracle may have the election fail, to simulate the case when a candidate does not get enough votes.

If it succeeds, it selects a free term number and records the server as leader in that term. It then chooses an uncommitted entry in the tree as the active node of the leader. The path to this entry will be the log of the leader. This way, we keep the fact that the leader always have all committed entries in its log.

2.3.2 Invocation

The invocation simply consists in adding a child to the leader's active node. The leader's active node then becomes this new node.

This operation simulates adding an entry to the leader's log. This entry is still uncommitted.

2.3.3 Push

The *push* operation allows a leader to commit some entries in its log (the branch to its active node).

Once again, instead of relying on communication with other servers, an oracle chooses a prefix of the active branch to be committed. This prefix can be empty, which means no entries are committed.

When entries are committed, their siblings are removed from the tree, since the committed part is supposed to be shared by all servers.

2.3.4 Reconfiguration

The ADORE[4] model is an extension of ADO supporting reconfiguration. As stated previously we are not interested in reconfiguration for now, but ADORE could replace ADO to prove that feature.

3 Refining Raft

3.1 Base layer

First of all, since we are not proving the Raft protocol itself but an implementation, we must choose this implementation. The main constraint is that it should be written in C, since the tools we use are made for this language. We picked a fairly simple, BSD licensed implementation of Raft[11]. It would of course be more interesting to prove a more widely used implementation, but we chose to keep things simple for the moment.

3.1.1 Details about the implementation

Raft is implemented as a library with functions to be called and callbacks to be implemented by the user. The user has to handle the communication, whether it is through the network or through any other mean.

The main functions for the user are the following:

raft_periodic The user has to call **raft_periodic** regularly, with the time elapsed since the last call as an argument. This function triggers the timeouts and the application of commands to the state machine.

raft_rcv_* There are a number of functions of that shape: **raft_rcv_requestvote**, **raft_rcv_requestvote_response**, and so on. Those functions must be called by the user when a Raft message is received, to give the contents of the message to the library. When relevant, those functions will give back a response message to be sent back.

raft_add_node There are functions for reconfiguration in the library, but those are out of the scope of the current work. However it is still necessary to give the initial configuration. The function **raft_add_node** allows to add a server to the configuration.

The callbacks are the following:

send_requestvote, send_appendentries Two callbacks asking the user to send the basic messages of the Raft algorithm.

log_* Several callbacks are used to give the user a copy of the log. The library has its internal version of the log, where commands are irrelevant since the library knows nothing about the state machine, but the user needs this log to be able to apply the commands to the state machine.

applylog This callback asks the user to apply the committed log entries to the state machine.

There are other callbacks for the snapshots, to store some data in persistent memory and more, but those are less important.

3.1.2 Getting CCR modules from C

In order to obtain a CCR module from C code, we used the Clight language provided by CompCert [1]. This language is a Coq module is a simplified version of CompCert C. CompCert provides a compiler from C to Clight, which we used to get a Clight version of the Raft library. This requires some minor editions to the original code:

- This compiler does not support functions with a variable number of arguments, so print functions with a fixed number of arguments have to be used instead of the standard ones.
- The original code is only divided into four files, some of which are long (up to around 1500 lines). In order to have more modularity, those files have been divided into smaller ones.

To define CCR modules from the Clight files, we use some Coq functions written by the team turning Clight code into Interaction Tree code. This IT code is then wrapped into a CCR module, making one CCR module per file.

3.2 Abstracting the C code

The first step in the refinement is to abstract the technical details introduced by C.

The most important step is to abstract away as many calls to memory as possible. Since CCR has a memory module, it is possible to access the state of the memory in the refinement proof, and thus have an abstract program replacing the memory with data structures in Coq. Some other technical details can also be abstracted, such as integer sizes. This obviously requires some hypotheses to be made about the integers in the code, but triggering UB (see definition 4) whenever an integer exceeds its bounds in the abstract code is enough.

A major difficulty with memory is the heavy use of C `struct` in the code. The way they are represented in Clight makes the computation of the offset for a specific field very inefficient. This makes doing interactive proofs impossible without additional tools.

3.2.1 Automation

My attempt at solving this was to define a first abstraction layer made of Interaction Tree code keeping as close as possible to the C code, but replacing memory accesses specifying the field name by the value of the pointer offset for that field. The idea is to have the computation heavy steps at this layer, to ease the proofs for the next layers.

The objective would be to have an almost automated proof for this refinement. I have spent a significant amount of time writing some tactics to try and achieve this goal. Unfortunately full automation seems out of reach for a lot of functions, especially when they involve loops or complex conditions. I have however been able to make it work on simpler functions. The computation time problem has not completely been solved, as some proofs still take a significant amount of time to finish, however this time has been reduced significantly and allows for interactive proving.

Another issue of this approach is that defining Interaction Tree code close to the C code is very time consuming. I have redefined almost all functions for this layer, however the refinement has not been proven for most.

3.2.2 Memory abstraction

The next layer of abstraction is the one removing most calls to memory functions. All calls cannot be removed for two reasons:

- The functions that the client can call are still expected to modify memory as a side effect. However we can read memory at the beginning of the function, use Coq data structures in the body, and store their contents at the end.

- We do not abstract callbacks (especially those related to communication) in this layer. Those functions must be called within a correct memory environment. We can do the same as with the previous point: store data in memory before calling and get it back after the call.

I have only partially implemented this layer, and no proofs have been done yet. It is however much faster to implement than the previous one. The refinement proof could be done with the previous layer, as initially planned, but it may actually be better to do the proof directly from the base layer, and ditch the previous layer altogether, by having the right lemmas linking the `struct` fields in `Clight` and those in the `Coq` data structures.

3.3 ADO layer

3.3.1 Structure of the layer

The last layer is made of ADO and some code to replicate the interface and additional behaviors from the Raft library.

Since ADO has only three operation (pull, invoke and pull), there has to be some code replicating the interface and behaviors of the Raft library, using ADO in the background. These behaviors include callbacks for example, since their implementation is left to the user and we don't know what side effects they can possibly have.

A key difference between this layer and the previous one is the fact that ADO does not use a communication channel (such as the network) but a global state. In order to share this state with the other servers in the abstraction, we can add an oracle to the `yield` function. The `yield` function is used to model concurrency, which will have to appear in this layer as it is needed to model communication between several instances. The ADO global state would be given to `yield` as an argument, and given back as a return value when the calling instance gets the priority back.

This layer has not been defined yet, but there is already an implementation of ADORE in `Coq` that we can use.

3.3.2 Ideas for the proof

Going from distributed states to a global one means that we have to take into account the communication between the servers. Since this communication is handled by the user code, and also involves other servers in the network, we have to use separation logic to add the relevant conditions and resources on the environment in which the library is used.

The conditions would be on how the user calls the functions. Using the library requires to add the necessary nodes to the configuration, call the `Raft_periodic` function regularly, provide appropriate arguments to the functions called...

On the communication side, resources can be used to model the messages. Sending a message creates a resource, which can be used to receive it (with a way to model lost and duplicated messages). We have to assume that other servers send correct messages (since Raft assumes there is no bad actor in the network).

3.4 Closing the proof

Once the top layer is achieved, we can extend the safety proof of ADO to show the safety of the top layer, which proves the safety of the base layer.

3.5 Summary of the layers

This is a summary of the structure of the proof. On the left is a server using the Raft library, and on the right is another server using an unknown implementation of Raft. $\text{Raft}_{\text{base}}$ is the base layer, and $\text{Raft}_{\text{abs mem}}$ is the layer with abstracted memory.

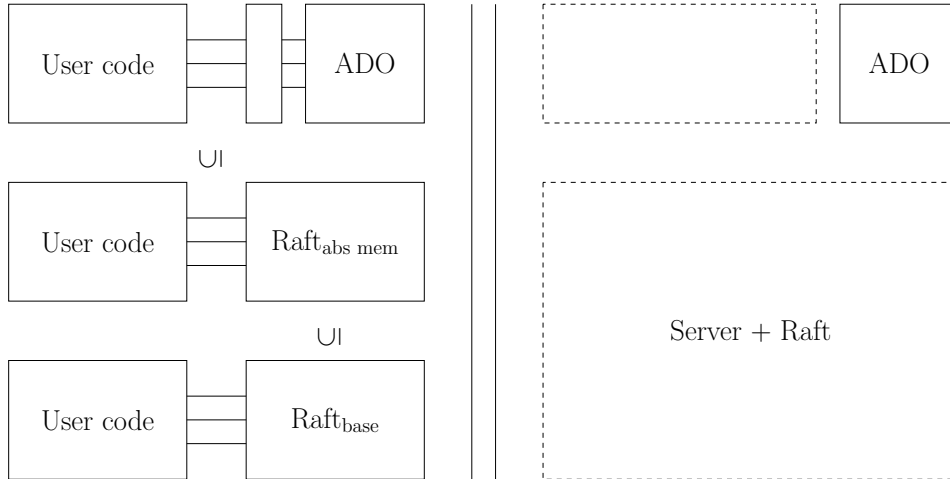


Figure 1: Summary of the structure of the proof

4 Proving a full implementation

The final objective is to be able to prove a full implementation using Raft.

4.1 Proving a single server

The proof of the library was made with an abstract user code. Hence proving that the code of a server outside of the library satisfies the conditions required for the proof of the last layer is enough to prove the safety of the full server regarding Raft.

4.2 Proving a network of servers

Once all the servers in a network (possibly using completely different implementations of Raft) are individually proven, we have to prove that the whole network of servers is safe. Proving them individually is not enough, because they may use different conventions for messages, or simply not use the same communication channel altogether.

Once again, since the other servers were abstract in our proof, it is enough to check the conditions on communication required for the proof of the last layer.

4.2.1 Network model

Proving that the way a communication channel is used satisfies certain properties requires a model of this communication channel. I have written a network model which can be used for this purpose, if the implementation communicates using network function.

The model is written as a CCR module, with the C socket functions (`socket`, `bind`, `connect`, `send...`) as interface. The interesting part of the model is the way messages are handled:

- When a socket sends a message, it is added to a list of messages on the receiving socket side.
- When a socket calls `recv` (to fetch messages it has received), a non-deterministic choice is made between returning no message, or one message in the list.

This satisfies the hypothesis of Paxos: a message may never be chosen, which simulates a lost message, may be chosen multiple times and may be reordered.

This module is rather low-level, as defining a more abstract version would be difficult due to the fact that functions each have a very specific effect. There are still a number of abstracted details: in most cases where a function should throw an error, the module triggers an undefined behavior. The verification should prove that these cases never happen.

This module was first used in the first part of the project, when working on Paxos. It allowed to extract the Paxos code to OCaml to test it.

5 Future work

The immediate work is to finish the proof of the current library.

As written previously, the library we are proving was chosen for its simplicity, but proving a more widely used Raft implementation would be more interesting.

Other members of the lab are also working on weak memory concurrency[2], a model of memory taking into accounts the possible reordering of memory operations performed by some optimisation. Integration of this model for CCR is being worked on, so this could be added to the verification.

6 Conclusion

The work to prove the Raft implementation is still very much in progress, since we have encountered many technical difficulties. However we have made progress using CCR on real world code, which should be useful for future projects.

References

- [1] Sandrine Blazy and Xavier Leroy. “Mechanized Semantics for the Clight Subset of the C Language”. In: *Journal of Automated Reasoning* 43.3 (Oct. 2009), pp. 263–288. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-009-9148-3. URL: <http://link.springer.com/10.1007/s10817-009-9148-3>.
- [2] Minki Cho et al. “Sequential Reasoning for Optimizing Compilers under Weak Memory Concurrency”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, June 9, 2022, pp. 213–228. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523718. URL: <https://dl.acm.org/doi/10.1145/3519939.3523718>.
- [3] Chris Hawblitzel et al. “IronFleet: Proving Safety and Liveness of Practical Distributed Systems”. In: *Communications of the ACM* 60.7 (June 26, 2017), pp. 83–92. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3068608. URL: <https://dl.acm.org/doi/10.1145/3068608>.
- [4] Wolf Honoré et al. “Adore: Atomic Distributed Objects with Certified Reconfiguration”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, June 9, 2022, pp. 379–394. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523444. URL: <https://doi.org/10.1145/3519939.3523444>.
- [5] Wolf Honoré et al. “Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems”. In: *Proceedings of the ACM on Programming Languages* 5 (OOPSLA Oct. 15, 2021), 97:1–97:31. DOI: 10.1145/3485474. URL: <https://doi.org/10.1145/3485474>.

- [6] Ralf Jung et al. “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’15. New York, NY, USA: Association for Computing Machinery, Jan. 14, 2015, pp. 637–650. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676980. URL: <https://doi.org/10.1145/2676726.2676980>.
- [7] Leslie Lamport. “Paxos Made Simple”. In: *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001) (Dec. 15, 2001), pp. 51–58. URL: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>.
- [8] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: (June 2014), pp. 305–319. URL: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>.
- [9] Ji-Yong Shin et al. “WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’19. New York, NY, USA: Association for Computing Machinery, Nov. 20, 2019, pp. 299–311. ISBN: 978-1-4503-6973-2. DOI: 10.1145/3357223.3362739. URL: <https://doi.org/10.1145/3357223.3362739>.
- [10] Youngju Song et al. “Conditional Contextual Refinement (CCR)”. Mar. 14, 2022. arXiv: 2203.07431 [cs]. URL: <http://arxiv.org/abs/2203.07431>.
- [11] Willem. *WillemT/Virtraft*. May 3, 2022. URL: <https://github.com/willemT/virtraft>.
- [12] Doug Woos et al. “Planning for Change in a Formal Verification of the Raft Consensus Protocol”. In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2016. New York, NY, USA: Association for Computing Machinery, Jan. 18, 2016, pp. 154–165. ISBN: 978-1-4503-4127-1. DOI: 10.1145/2854065.2854081. URL: <https://dl.acm.org/doi/10.1145/2854065.2854081>.
- [13] Li-yao Xia et al. “Interaction Trees: Representing Recursive and Impure Programs in Coq”. In: *Proceedings of the ACM on Programming Languages* 4 (POPL Jan. 2020), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3371119. URL: <https://dl.acm.org/doi/10.1145/3371119>.