Kalpa Publications in Computing
Volume 20, 2024, Pages 59–73
Proceedings of 2024 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference

Building Towards a Distributed, Dynamic Solution to the Santa Problem

Mahmood Mohammed Seoud[1] and David Gray Marchant[2]

[1] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
ORCiD: 0009-0002-7638-8892, mahmoodseoud@gmail.com
[2] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
ORCiD: 0000-0003-4262-7138, david.marchant@di.ku.dk

Abstract

In this paper, we delve into the creation of a mesh grid network, focusing on solving the Santa Claus Problem in concurrent programming. It discusses how we implemented a client/server communication model and tested the system's reliability and robustness. Our findings have significant implications for the advancement of wireless communication technologies, especially Bluetooth mesh grids, and point to the necessity of future work involving actual distributed hardware for validation.

1 Introduction

This paper focuses on designing an adaptive network protocol capable of seamlessly accommodating the dynamic nature of distributed network devices as they join and leave the network. While this solution does not achieve full dynamism, it addresses crucial aspects necessary for future development. By investigating key elements such as dynamic robustness, system correctness, and overall performance.

2 Background

2.1 The Santa Clause Problem

The Santa Clause problem is stated as follows:
"Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elfs problem is never serious enough to wake up Santa (Otherwise, he may never get any sleep), so, the elves visits Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those. If Santa wakes up to find three elves waiting at his shops door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefor they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year

in paradise this could also explain the quickness in their delivering of presents, since the reindeer cant wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh."[1]

2.2 Communicating Sequential Processes (CSP)

Creating a dynamic distributed systempresents significant challenges due to the inherent complexity of such networks. Nodes can join and leave the network at will. This level of flexibility and peer-to-peer connectivity, while advantageous for certain applications, introduces substantial difficulties in terms of network management and stability.

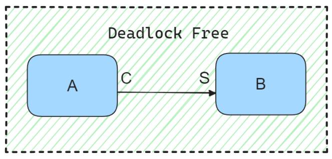
One effective way to approach the design of such a distributed system is by leveraging the paradigm of Communicating Sequential Processes (CSP)[2]. CSP, introduced by Tony Hoare in 1978, is a formal language for describing patterns of interaction in concurrent systems. It provides a framework for structuring and managing concurrent processes, making it a suitable choice for designing dynamic distributed systems.

CSP models concurrent systems as a collection of independent processes that communicate through well-defined channels. The structured method of CSP simplifies the management of concurrent tasks and enhances the predictability and reliability of the system.

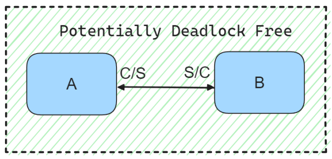
In the context of designing a dynamic distributed system, CSP provides a solid foundation for managing concurrency and ensuring robust communication between nodes by offering protection against deadlocks and race conditions. By adopting CSP, the complexity of the system can be reduced, making it easier to implement and maintain. This approach not only simplifies the development process but also enhances the system's ability to handle dynamic changes, ensuring reliable performance and correctness in a distributed environment.

2.3 Clients and Servers

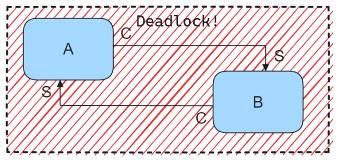
When designing a parallel system, it's crucial to ensure its proper functionality and absence of issues. In concurrent systems, such as those employing the CSP model, implementing a client/server structure can effectively address common problems like deadlocks.



(a) Simple version of client-server communication. Deadlock-free.



(b) Two clients and two servers that operate in opposite pairs. Potentially deadlock-free.



(c) Two clients and two servers at different times. Not deadlock-free.

Figure 1: Client/Server examples. Note that communication can go both ways in all 3 examples

In Figure 1 we see a presentation of three simple systems. The first example in Figure 1a shows two processes – a client and a server. This setup is simple and avoids deadlock. The second example in Figure 1b shows a server responding to a client. Notice that both ends can be either server or client, but in this example, they will always act opposite of each other and therefore also avoid deadlock. However, the third example in Figure 1c presents a situation to avoid. Two processes try to function as clients at different times. If they both do this simultaneously, deadlock becomes a possibility. This situation happens when the process interactions form a loop. By analyzing process interactions, we can identify if the system is deadlock-free, especially in complex problems such as the Santa Claus Problem. When client and server communicate without loops, it avoids deadlock. However, even if loops are present, the system can avoid deadlock if the interactions are managed well.

2.4 Background Literature

Communicating Sequential Processes for Java (`JCSP`)[3] and Communicating Sequential Processes (`pycsp`)[4] are libraries that implement the CSP paradigm for concurrent programming. These libraries provide tools to build systems where processes communicate with one another via channels while adhering to the principles of CSP. Both libraries are no longer actively maintained and therefore, have not been used in this project.

Trono wrote an implementation for the Santa Clause Problem in the language C using threads to distribute work and semaphores to synchronize it [1]. This approach utilized shared memory to declare and update variables that, for instance, could keep count of how many Reindeer had arrived, and the last one woke up Santa. Unfortunately, the solution of utilizing Trono is not feasible for a dynamic distributed system, as it relies on shared memory between threads and is therefore not compatible with network usage.

A study conducted by Jason Hurt and Jan B. Pedersen [5] examined several methods of writing parallel programs to solve the Santa Clause Problem, and evaluated the advantages and disadvantages of each approach. They explored three different models: the popular shared memory model (threads), a message-passing interface (`MPI`)[6], and a process-oriented approach using `JCSP`. None of these shall be used within this project, as they either use no longer supported implementations or lack inbuilt support for forming consensus across a network.

A distributed solution to the Santa Problem has been previously demonstrated by David Gray Marchant[7]. However, the solution cannot be directly copied as it assumes a fixed number of elves and reindeer, which does not fit my scenario where the number of elves and reindeer is dynamic. However there is a design choice, that is usable. Mainly the ”chain”. More on what parts the chain plays in this project in section 4.

2.5 Paxos

The Paxos algorithm[8] makes it possible to reach an agreement or consensus, even when there are issues like failures and asynchrony. The protocol is built around three roles: proposers, acceptors, and learners. Proposers suggest values, acceptors decide whether to accept or reject them based on specific criteria and learners observe the process and eventually agree on the chosen value. The Paxos protocol guarantees safety and liveness properties, ensuring that agreed-upon values are valid and eventually chosen.

Although Paxos is theoretically sound, it can be challenging to implement and understand. The protocol's intricacies and edge cases often make it difficult to grasp, leading to pitfalls and inefficiencies in real-world deployments [8].

2.6 Raft

Raft offers a more intuitive consensus approach compared to Paxos, which is known for its complexity. Raft divides the consensus process into three phases: leader election, log replication, and safety properties.

Central to Raft is the leader node, which manages consensus and coordinates state changes across the cluster. The leader election mechanism ensures that only one node serves as the leader at any time. In case of leader failure or network partition, Raft uses a straightforward election protocol to maintain continuity and fault tolerance. Nodes start in the follower state, responding to leader requests and voting during elections. They monitor heartbeat messages to track the leader's presence. If a follower misses a heartbeat, it transitions to the candidate state, initiating a new election term, and requesting votes from other nodes. If a candidate receives votes from a majority, it becomes the leader. If no majority is reached or multiple candidates receive votes, a new term begins, and the election process repeats. The different states a Raft cluster node can go through are depicted in Figure 2

Once a leader is established, it oversees log replication—a sequential record of state changes across all nodes. Through append entry Remote Procedure Calls (RPCs), the leader disseminates log entries to followers, who replicate these entries verbatim. Raft ensures consistency by committing log entries once a majority of nodes acknowledge them.

Raft upholds several safety properties, including leader completeness and election safety, to maintain consensus integrity even during failures or partitions. Leader completeness ensures all committed log entries are present on the leader.

Nodes transition between follower, candidate, and leader states due to factors like network leader failures or timeouts. Raft enables fault-tolerant distributed systems, allowing data replication across multiple servers.

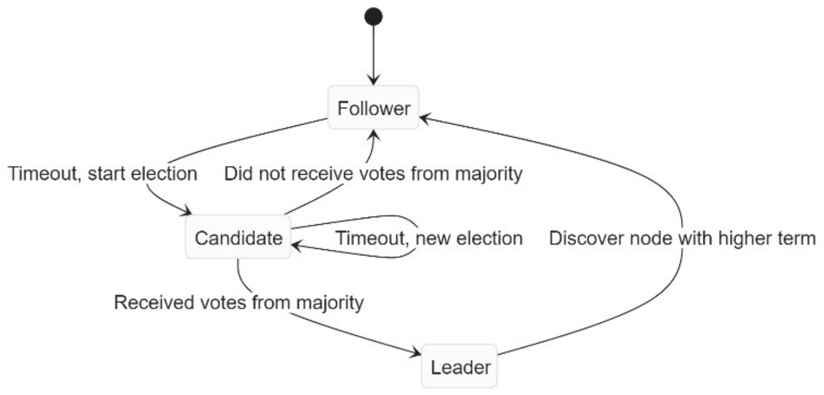

Follower
Timeout, start election
Did not receive votes from majority
Candidate
Timeout, new election
Discover node with higher term
Received votes from majority
Leader

Figure 2: Raft diagram

While Raft and Paxos offer robust consensus solutions for distributed systems, they primarily target static networks. Dynamic networks, characterized by node mobility, intermittent connectivity, and varying signal strengths, require methods to add nodes without disruption. This is discussed further in section 4.2.

Raft was chosen for its simplicity and the availability of robust libraries and frameworks, such as `PySyncObj`[9], more on this in section 3.2.1.

2.7 Requirements for a Dynamic Distributed System

To develop a dynamic distributed solution for the Santa Claus Problem, it is important to establish specific system requirements. These can be seen in Table 1. Some of these requirements come from the original problem and others are inspired by various solutions to the distributed Santa Clause Problem like the one from Marchant [7].

The system must also be dynamic, meaning that it should be able to handle:

Requirement	Description
1	The system should have one Santa thread.
2	The system should start with nine Reindeer threads.
3	The system should start with ten Elf threads.
4	All instances of Elves must be running the same code as all other instances of Elves.
5	All instances of Reindeer must be running the same code as all other instances of Reindeer.
6	Santa must be able to sleep and only sleep when not consulting Elves or leading Reindeer.
7	Santa should only be woken once all Reindeer or three Elves are ready to deliver presents/consult.
8	Santa must always prioritize delivering presents with Reindeer rather than consulting with Elves unless a consultation is already in progress, in which case the Reindeer will wait until it is over.
9	The Reindeer must go on holiday, each for a random amount of time.
10	The last Reindeer to arrive must wake up Santa.
11	The Elves must get to work in the workshop, each for a random amount of time (handled by consensus algorithm).
12	Only 3 Elves at a time should consult with Santa about toys.
13	While a group of three Elves consult Santa, the other elves should be able to create their groups.
14	If there are two groups of Elves looking to talk to Santa, the first to arrive should be served first by Santa.

Table 1: System Requirements for a distributed system

- Any node unexpectedly losing connection.
- Re-connections or previously known nodes.
- Introduction of new nodes.

3 Setup

3.1 Hardware

An Apple MacBook Air 2020 M1 with 8 GB of memory and 8 cores was utilized to execute the solution, running the latest version of macOS Ventura. The communication was managed through socket communication using the AF_INET socket family[1]. It is important to clarify that no actual network communication was utilized. However, a multi-processed prototype system was run locally on the above machine in a way that could be easily replicated over a network. The decision to use a local network helped minimize the chances of firewalls, devices losing connection, or any other issues disrupting communication.

[1]AF_INET address family is represented by A pair (host, port)

3.2 Software

The software in this project was written in Python. This was due to the authors already being familiar with it and the benefits of Python's very simple syntax. Moreover, Python provides modules that are easy to use such as `socket` and `socketServer` to communicate between servers which is the primary source of communication in this project. Furthermore, Python has very good libraries available to it such as `PySyncObj`. This is a Python library for building fault-tolerant distributed systems. It provides the ability to replicate your application data between multiple servers. Thus Python is an appropriate language to solve the Santa Claus Problem.

In this software, we utilized Python's threading module, which operates with the Global Interpreter Lock (GIL) [10]. The GIL allows only one thread to execute Python code at a time, which can result in reduced performance. This issue can be mitigated by using the multiprocessing module instead. Nonetheless, threading remains an appropriate choice for this project because performance is not a critical concern. The source code for the project can be found here [11].

3.2.1 `PySyncObj` (Raft implementation)

As mentioned, the authors selected Raft as the consensus algorithm for this project. The `PySyncObj` library implements Raft and provides essential methods and features. In Table 2, you can find a brief overview of the functions used in the library.

Function/Method	**Explanation**
`syncobj_admin`	A utility for monitoring the cluster state and adding or removing cluster nodes, useful during the testing stages.
`addNodeToCluster`	This method notifies all nodes in the cluster of a new addition and provides the new node with a list of current cluster members, ensuring it has the necessary information.
`isNodeConnected`	Allows the system to verify the connection state of nodes, checking if any members have disconnected during interactions.
`ReplLockManager(autoUnlockTime)`	A distributed lock that ensures that if any node disconnects while holding a lock, it will automatically release the lock after the specified `autoUnlockTime`. For this paper, the `autoUnlockTime` is set to 10 seconds.

Table 2: Some functions and Methods in `PySyncObj` Library

4 Implementation

4.1 Design

To create a dynamic solution, it was decided to break down the problem into three main sections; Santa, Reindeer and Elves. This design choice was heavily inspired by Marchant [7]. Each Reindeer or Elf should be able to communicate with Santa and also should be able to communicate with all the other Reindeer or Elves respectively. This gives us the following overall structure as seen in Figure 3. It is important to note that the chain is not a separate

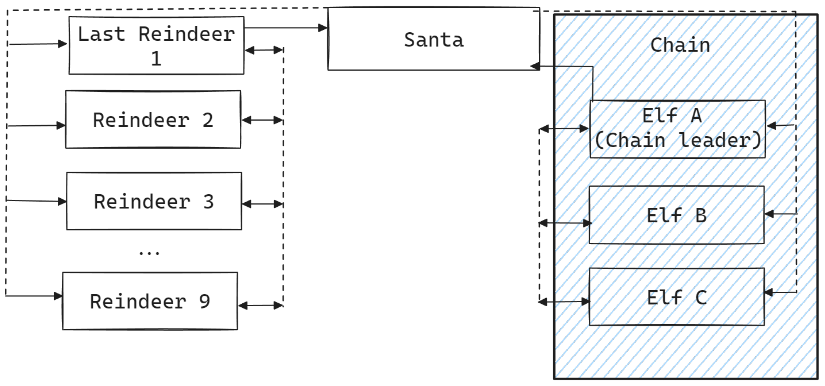


Figure 3: The overall solution architecture for the distributed system involves the interaction of Reindeer and Elves, which can occur in any order. This means that Reindeer 1 is not always the last and there could be a different order. The same applies to the chain, which is a subset of the Elves when there are more than 3.

process; it exists solely as a list for the chain leader. In terms of communication, the last Reindeer sends a message to Santa, who then relays the message to all the Reindeer. For the Elves, the chain leader sends a message to Santa, who then relays the message to all chain members. Please refer to Figure 4 for details about the Santa diagram, Figure 5 for details about the Reindeer, and Figure 7 for details about the Elves.

4.1.1 Santa

Santa's design is quite straightforward, as he is only required to listen to messages from either Elves or Reindeer (as stated in requirement 7 of Table 1). Santa's listening capability can be implemented using a single thread. Although the thread can receive messages from both Elves and Reindeer, Santa must be able to differentiate between them (as stated in requirement 8) and prioritize Reindeer-related tasks over those related to Elves. This differentiation is achieved by having both Elves and Reindeer provide an identifier, which Santa can use to distinguish between them and execute Reindeer-related tasks before those related to Elves, using an if-else condition. The states Santa be in with this approach are illustrated in Figure 4. This setup follows the design in Figure 1a which means that Santa acts as a server and never initiates any communication on his own. Thus, Santa will never be deadlocked.

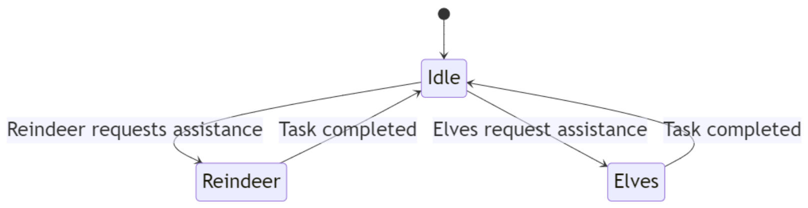


Figure 4: Santa network state diagram.

4.1.2 Reindeer

Reindeer are all connected since they participate in a cluster. A depiction of this can be seen in Figure 5. Each Reindeer has two sub-components and one that coordinates them. The

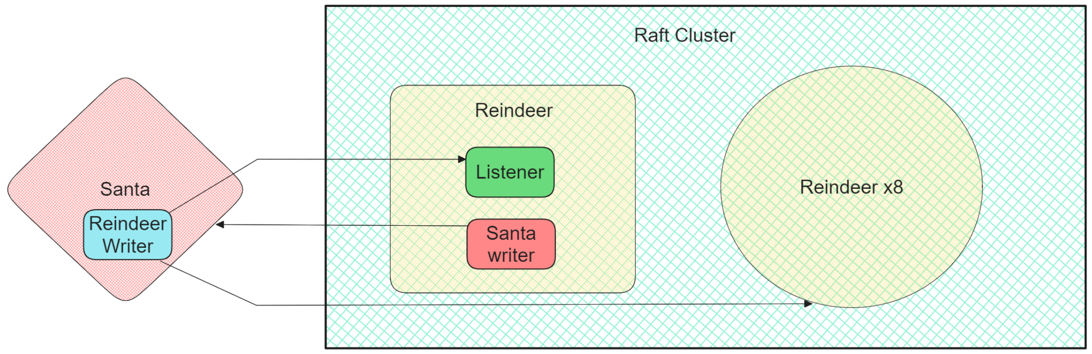


Figure 5: Reindeer architecture. Note that there are eight other Reindeer, which means that there are equally many listeners and Santa writers. The Santa writer never interacts with other processes than Santa.

first listens to incoming messages, and the other sends messages to Santa. Once a Reindeer has contacted Santa he will contact all Reindeer. A Reindeer will thus either initiate communication or receive it, and sometimes even both. This means that we need to be careful with how this is executed to avoid deadlock, referring back to Figure 1b. The Reindeer begins its holiday by sleeping for a random amount of seconds (between one to five seconds). When a Reindeer wakes up they will add itself and its sleep time to a shared set. Once all the Reindeer have woken up, they determine the last Reindeer to wake up amongst them by finding the maximum sleep time in the set. If two Reindeer happen to have the same sleep time, the first occurrence will be chosen due to how the *max* function works in Python. However, this is not an issue since the distributed set is the same across all Reindeer processes, so all Reindeer will always agree on who the last Reindeer to wake is. After determining the last Reindeer to wake up, it will contact Santa while the other Reindeer wait for Santa's response. The last Reindeer provides Santa with each Reindeer's port to contact. After Santa responds, the whole process starts over again.

4.1.3 Elves

The system for distributing the Elves follows a client-server model using locks and Raft. Unlike the Reindeer, which can wait for all members to arrive, the Elves need to wait for a dynamically changing subgroup of the Elves. To model this, we have adopted the concept of a chain, inspired by Marchant [7]. Once an Elf experiences a problem, he attempts to acquire a distributed lock. If it succeeds, it adds itself to the distributed set chain. If it fails to acquire the lock on the first try, it will keep trying until it does. If it hasn't succeeded within 10 seconds, it will assume the connection is dropped and the lock will be released automatically. Once the chain has three members, the chain leader starts two threads - one for messaging Santa and another for listening for incoming messages. This would ensure that he is ready to receive incoming information from Santa. The remaining chain members only listen for incoming messages from Santa like in Figure 1a. For more details, see the state flow model for an Elf, which is shown in Figure 7.

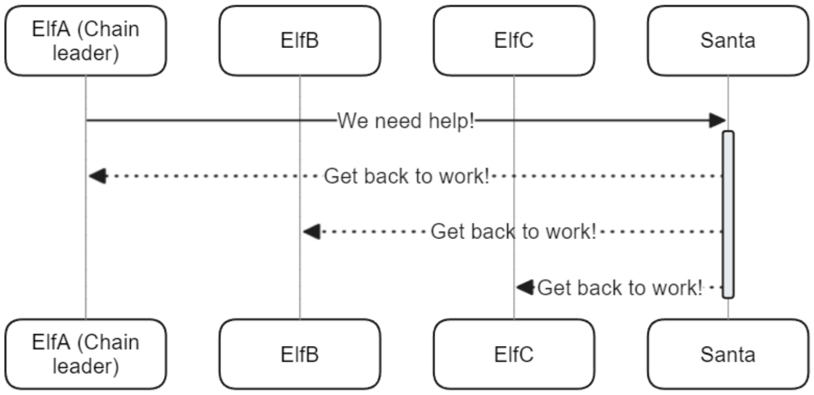


Figure 6: Santa's communication with the Elves. Santa is "idle" until he receives a message from ElfA.

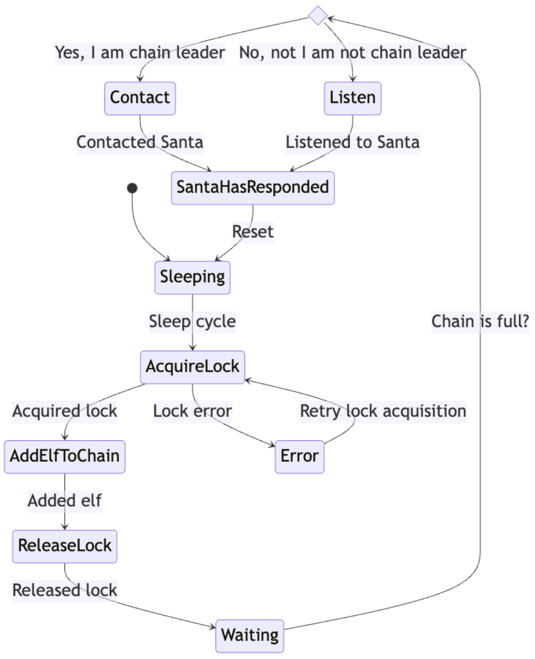


Figure 7: Elf state diagram

4.2 Adding Dynamism

The system previously described can be distributed, but is not yet dynamic. To do so, it must accommodate nodes leaving, rejoining, or joining for the first time. As the network communications are extensive, as are the ways they could be interrupted, this paper will now examine some illustrative examples, demonstrating a dynamic solution.

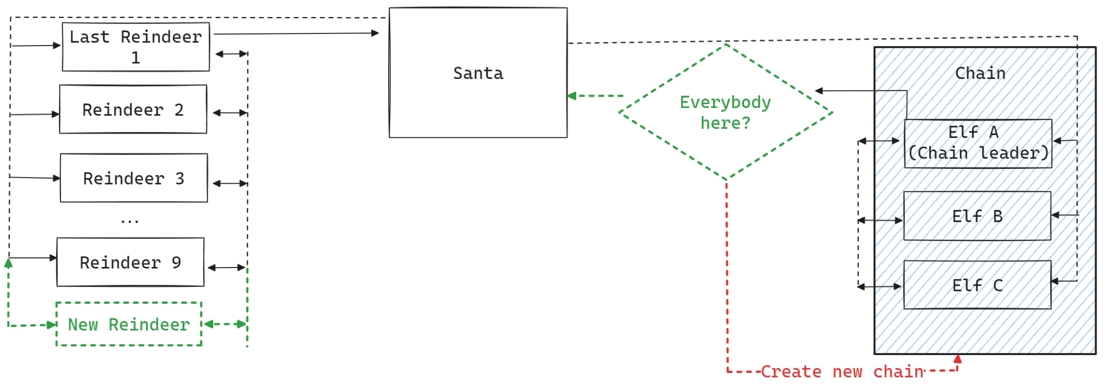


Figure 8: Full Dynamic Distributed System diagram. You can find the details for the additions to the Reindeer in this Figure 10 and in section 4.2.3. You can find the details for the additions to the Elves in section 4.2.1.

4.2.1 Loss of connection

In case of any disconnect, a strategy was needed to handle it. There are specific scenarios that will be examined. The first scenario is when a Node unexpectedly leaves the network in the time interval between a group forming (Either a chain or all reindeer) and the group leader reaching out to Santa. This would cause Santa to deadlock trying to contact the now missing Node, as illustrated in Figure 9 with the Elves.

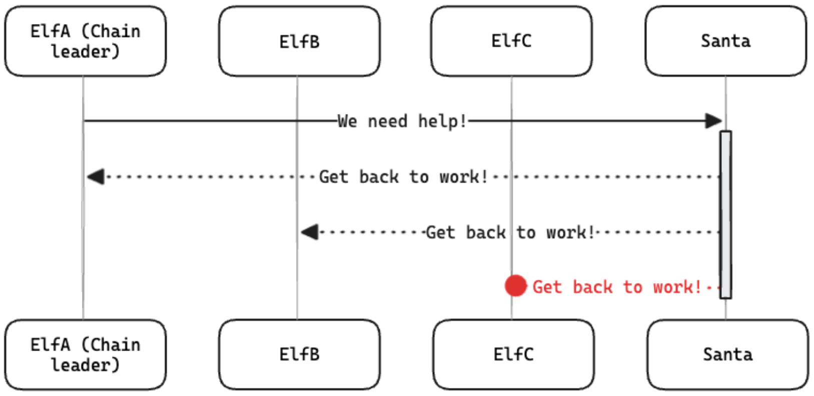


Figure 9: ElfC losing connection before receiving acknowledgment from Santa

To address this problem we need to make sure the group is entirely connected before reaching out to Santa. This is achievable because of a feature in the Raft implementation *PySyncObj* [9] called *isNodeConnected*, as presented in Table 2. It can thus check if any of the members have disconnected midway through any interactions. This does however not solve the issue completely but rather puts off the problem for a bit. Let's consider the event that an Elf happens to disconnect after this check but before Santa has responded. This would still make Santa crash regardless of the prior check. To address that, we can add a try-catch block around Santa's communication to prevent the program from crashing. However, this may contradict requirement 7 because now only 2 Elves visit Santa, which is not ideal but sufficient for the scope of this paper.

Another scenario to address is the one where a Node carrying a distributed lock happens to disconnect. The locks have an `autoUnlockTime` of 10 seconds, as shown in Table 2. This means that if any node disconnects while holding a lock, it will automatically release the lock after 10 seconds, preventing potential system issues caused by the disconnection.

4.2.2 Re-connection

If this system can handle the loss of connections, it must also handle re-connection. Fortunately, `PySyncObj` makes this very simple. They just need to restart. However, one limitation is that the node needs to know the entire cluster before reconnecting. This is not that big of a problem, since the node already was included in the cluster before disconnecting. But this approach is not perfect because if the cluster changed while the node was away, it may not start up with the correct cluster member list.

4.2.3 Introduction of new nodes

To introduce new nodes into the system, traditional client/server communication was utilized. Every Reindeer node has a listening port, such as is shown in Figure 5. This setup allows the new node to contact one of the existing nodes immediately. The existing node then adds the new node to the cluster using the `addNodeToCluster` method from the PySyncObj library, as presented in Table 2.

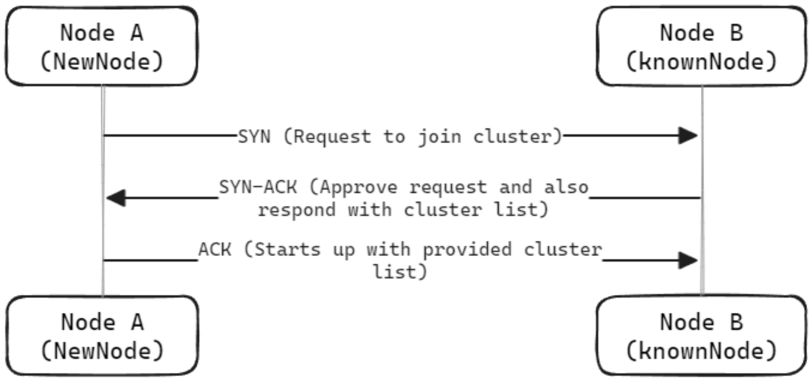
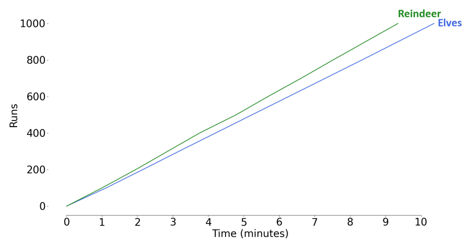

Node A (NewNode)
Node B (knownNode)
SYN (Request to join cluster)
SYN-ACK (Approve request and also respond with cluster list)
ACK (Starts up with provided cluster list)
Node A (NewNode)
Node B (knownNode)

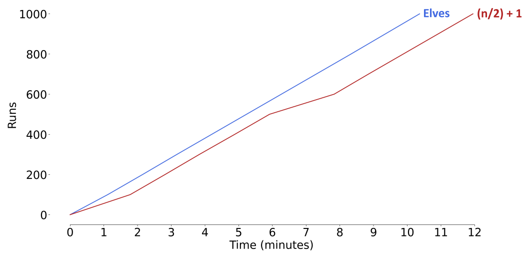
Figure 10: 3-way-Handshake for introduction of new Reindeer.

5 Testing the Dynamic Distributed Solution

In order to test the reliability of the solution, the authors ran 1 Santa process, 9 Reindeer processes, and 10 Elf processes, each on their terminal. Santa will log out messages to a file to indicate when he is consulting the Elves or working with the Reindeer. Each Elf and Reindeer will log their current states to a file, including when they acquire the lock, release it, add themselves to a shared set, or experience any errors. This logging system ensures a clear log history of all activities, providing a clear overview of the system's correctness.



(a) 9 Elves compared 10 Reindeer performance tests. The Reindeer completed 1000 runs in about 9 minutes The Elves completed their 1000 runs in about 10 minutes.



(b) 10 Elves compared to 6 Elves. It took the 10 Elves about 10 minutes to reach 1000 runs. It took the 6 Elves about 12 minutes to reach 1000 runs.

Figure 11: Performance comparison between different groups. 11a Shows the performance between Elves and Reindeer, with Reindeer being faster. 11b Shows the effect of reducing the number of Elves from 10 to 6, resulting in slower performance.

5.1 Deadlocks

As a crude test for deadlocks, the system was simply run for an extended period of time as Elves along with Santa, Reindeer alone with Santa, and then all three in combination. The tests were run overnight for over 16 hours and 42 minutes for the Elves (see log.zip). The same procedure for the Reindeer and it got similar results. Lastly, the system as a whole was tested for about 2 hours. However, it's important to note that running the system for 2 hours may not be rigorous enough, and rare deadlocks could go undetected. All tests were completed without encountering a deadlock.

5.2 Performance

For the testing of the system's performance, the number of times Santa had to consult the Elves or Reindeer and the timestamp at which he did so were traced. This was done separately for the Elves and the Reindeer. For these tests, any kind of sleep was removed to keep the testing controlled and focused on only performance of the solution.

For the distributed solution, as shown in Figure 11a, the Reindeer completed 1000 runs in about 9 minutes. This means the Reindeer has an average run time of approximately 0.5 seconds/run. For the distributed solution, the Elves completed their 1000 runs in just over 10 minutes, resulting in a slightly worse average time of 0.6 seconds/run.

Tests were then run on the Elves but with pre-determined disconnections on certain nodes to a reduced network down to $(n/2) + 1$ nodes, which is the minimal number of nodes Raft needs to function, to see how that would affect the performance of the network, see Figure 11b. In this case, 10 Elves started but they were reduced to 6. This setup made 1000 runs in 12 minutes.

5.3 Dynamic robustness

To test the robustness of the dynamic solution, some pre-selected processes were killed in attempt to interrupt the system. This was done by incorporating a `sys.exit()` for the selected

Elves. The processes were then observed through the logs and the `syncobj_admin` tool, specifically examining how the system behaved after this termination. The solution proved to be resilient and all remaining processes were able to progress without issue. We also evaluated whether the `autoUnlockTime` setting effectively prevented race conditions or deadlocks. For our tests, we configured it to 10 seconds. Given that the Elves in this project could potentially work for up to 5 seconds, the `autoUnlockTime` must be set to at least this duration to ensure they have sufficient time to complete their tasks. This precaution helps avoid false positives where the system might incorrectly assume disconnection before the Elves have finished their work.

6 Discussion

6.1 Performance

In section 5, it was noted that the entire system took just over 7 minutes to complete 1000 runs. It's not surprising that the processing is faster compared to either the Reindeer or Elves only. This is because more nodes are contributing to the runs. This run is significantly longer than the non-distributed system that completed the task in under 1 second. This is due to the overhead added by the sockets. The non-distributed system can bypass this completely. However, considering the objectives of this thesis, speed is not the top priority. This is primarily because when connecting systems over the internet, latency becomes a major concern rather than the actual hardware or software performance. Additionally, humans are much slower compared to computers, so a solution with an average run time of about 0.5 seconds is unlikely to be noticeable or cause annoyance from a human-computer interaction (HCI) perspective. This makes the performance acceptable, even though it took 7 minutes to reach 1000 runs.

In a situation where half of the network disconnects simultaneously and doesn't reconnect, we observed a decrease in performance from 10 minutes to 12 minutes. This suggests that the performance doesn't increase in proportion to the number of Elves. If it did, we would expect the half network to take about 2 times longer to finish. This performance issue could be a concern in a real Bluetooth mesh grid network. The reason for this could be that Santa handles incoming messages simultaneously.

6.2 Limitation

The Raft algorithm was generally a good choice in hindsight, as it provided good fault tolerance and performance. One limitation is that it requires $(n/2)+1$ nodes, where n is the total number of nodes in the network, to function correctly.

Despite the neat features of the Raft algorithm, the `PySyncObj` implementation, and its ability to replicate application data across multiple servers, it presented some challenges. The most challenging aspect was the lack of auto-discovery of new nodes, which would have made adding new nodes much easier. It's worth noting that this feature is currently being discussed in several GitHub issues on the `PySyncObj` repository and may be included in the library in the future [9]. Another challenge with `PySyncObj` is that it's not possible to remove the leader from the cluster. We can disconnect the leader, but we cannot remove them from the cluster. When a node is disconnected, it only means that the node is offline. The other cluster members still consider it part of the cluster. If you remove them from the cluster, they won't be considered part of it anymore. The last challenge that we want to address, regarding `PySyncObj`, is it does not allow for merging two partitioned networks into one large network, which is a feature of

Raft the algorithm. It would have been helpful if the chain could have its partition and then rejoin the larger network after consulting with Santa.

7 Future work

In planning for future work, there are several areas that can be explored to further develop and implement the findings of this paper. Firstly, the current implementation does not address the loss of connection for the Reindeer. By using similar principles as the done with the Elves this would only require minor changes to the Reindeer. Additionally, a re-connection mechanism for the Reindeer needs to be implemented. Secondly, the current system lacks support for the dynamic introduction of new nodes for the Elves. This will involve implementing a constant listener for the TCP-3Way handshake, requiring further work. Furthermore, conducting additional tests by increasing the total number of processes running will provide valuable insights into the system's capacity and performance under more demanding conditions. Finally the transition from using a simulated environment, locally, to actual networked communication on real distributed hardware. This will allow for more realistic testing and validation of the system's functionality in real-world conditions.

8 Conclusion

This paper discussed the development and implementation of an adaptive network protocol that can accommodate the dynamic nature of nodes joining and leaving the network, despite limitations in the `PySyncObj` implementation. This paper also examined various aspects such as dynamic robustness, system correctness, and overall performance of the network. This paper has the potential to significantly contribute to the advancements in wireless communication technologies, particularly Bluetooth mesh grids, which can be further developed to meet modern network requirements.

References

[1] J. Trono, "A new exercise in concurrency," *St . Michael's College*, 1994. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/187387.187391.

[2] T. Hoare, "Communicating sequential processes," 1985. [Online]. Available: http://www.usingcsp.com/cspbook.pdf.

[3] P. Welch and N. Brown, "Communicating sequential processes for javatm (jcsp)," 2014. [Online]. Available: https://www.cs.kent.ac.uk/projects/ofa/jcsp.

[4] J. Bjørndalen, B. Vinter, and O. Anshus, "Pycsp - communicating sequential processes for python.," vol. 65, Jan. 2007, pp. 229–248.

[5] J. Hurt and J. B. Pedersen, "Solving the santa claus problem: A comparison of various concurrent programming techniques," *School of Computer Science, University of Nevada*, 2008. [Online]. Available: https://www.researchgate.net/publication/221004442_Solving_the_Santa_Claus_Problem_a_Comparison_of_Various_Concurrent_Programming_Techniques.

[6] A. S. Gillis, "Message passing interface (mpi)," 2022. [Online]. Available: https://www.techtarget.com/searchenterprisedesktop/definition/message-passing-interface-MPI.

[7] D. Marchant and J. Kerridge, "Solving the santa claus problem over a distributed system," *Niels Bohr Institute, University of Copenhage School of Computing, Edinburgh Napier University*, 2018. [Online]. Available: https://www.napier.ac.uk/~/media/worktribe/output-1305147/solving-the-santa-claus-problem-over-a-distributed-system.pdf.

[8] L. Lambort, "Paxos definition," 1989. [Online]. Available: https://www.scylladb.com/glossary/paxos-consensus-algorithm.

[9] F. Ozinov, "Pysyncobj," 2016. [Online]. Available: https://github.com/bakwc/PySyncObj.

[10] P. S. Foundation, "Threading — thread-based parallelism," 2024. [Online]. Available: https://docs.python.org/3/library/threading.html.

[11] M. M. Seoud and D. G. Marchant, "Github repo for bachelor thesis," 2024. [Online]. Available: https://github.com/MahmoodSeoud/BachelorThesis.