

Solving the Santa Claus Problem Over a Distributed System¹

David MARCHANT^{a,1} and Jon KERRIDGE^b

^a*Niels Bohr Institute, University of Copenhagen*

^b*School of Computing, Edinburgh Napier University*

Abstract. A working solution to the Santa Claus Problem is demonstrated that operates over a distributed system. It is designed and modelled using the client/server model to maintain a deadlock and livelock free architecture. The system was developed using the JCSP library for Java and demonstrated over a network of twenty desktop PCs. Each Santa, Reindeer and Elf process is broken down into numerous sub-processes, and are each designed according to the client/server model. A novel approach to the Elves is presented using a chain-based architecture to allow non-determined sub-groups to communicate and consult with Santa in sets of three.

Keywords. Santa Claus Problem, CSP, JCSP, Client/Server model

Introduction

The Santa Claus Problem was first proposed as a way of introducing new students to the problems of concurrent and parallel programming, and so is intended as a reasonably simple problem to solve on a single machine running multiple concurrent processes. This paper will look at a distributed solution running on multiple machines in parallel. Different PCs will be used to represent the various processes with the aim that they can interact with each other to demonstrate a solution. For the system to be judged as a success it must meet the requirements set out by Trono's original proposition, and must be deadlock, livelock and error free. Freedom from deadlock and livelock can be guaranteed by correct application of the client/server model so this shall be used throughout the design. The client/server model is needed as there are many additional challenges in distributed systems [1].

1. Background

1.1. Problem Definition

The Santa Claus Problem is an exercise in designing concurrent and parallel systems. It was initially proposed by John Trono in his paper *A New Exercise in Concurrency* [2]. The problem states:

“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 elf's problem is never serious enough to wake up Santa (Otherwise, he may never get any sleep), so, the elves visit 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

¹Corresponding Author: *David Marchant, Niels Bohr Institute, University of Copenhagen, Denmark*. E-mail: d.marchant@ed-alumni.net.

elves to return. 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 therefore 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 can't 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. Motivation

In the years since Trono first proposed his problem, concurrent and parallel systems have become possible over networks. The simple problem that Trono proposes may now serve as an introduction to the additional problems presented by these distributed systems. Problems such as a lack of global resources mean that a great deal of solutions to the Santa Claus Problem are not applicable to a distributed solution, as shown in Section 1.3. For this reason, this paper shall explore a new solution specifically for a distributed system.

1.3. Background Literature

A New Exercise in Concurrency [2] is the starting point for the paper as it is where the Santa Claus Problem was first proposed. Trono talks about his desire for a new exercise to be used to help teach core concepts of parallel processing and so comes up with his problem. He then proposes a solution to the problem and considers it solved within a 3-page paper. His solution relies on semaphores and shared memory to count the number of Reindeer and Elves that have arrived. This is sufficient for a small exercise, but shared memory is impossible over a network such as between several independent machines. For this reason we will need a different solution. Additionally, Santa must check each time a Reindeer arrives to see if that is the last Reindeer. This directly contradicts the last sentence of his initial proposal and so it is possible that his solution is not really a solution at all.

Trono's solution is rejected in the paper *How to solve the Santa Claus problem* [3], though on different grounds. Ben-Ari states that semaphores are not sufficient as it assumes that any process released from waiting will automatically be scheduled again. He then states that there are semaphore algorithms that could solve this issue, but that they are too complex for such an introductory exercise. A solution in Ada 95 is then proposed, which uses barriers to count in processes. These barriers are in external 'room' processes, with separate rooms for Reindeer and Elves. Though Ben-Ari does not refer to them as such, the room for the Reindeer shall be referred to as the stable and the vestibule for the Elves. This is to make the solution more consistent with later discussed solutions. In the case of the stable it will count in the Reindeer, and only wake Santa once all are assembled. In the case of the Elves it will wait for three before waking Santa. Like Trono's solution, this also ignores the last sentence, as the last Reindeer is not the one to wake Santa, it is the stable. This problem is discussed and solved in Ben-Ari's paper but there is still the use of stables and vestibules which seems an unsatisfying solution. Trono's problem makes no mention of external processes. A true solution should have the processes being able to count themselves in and identify who else has already arrived without the aid of some external barriers.

In the paper *Santa Claus: Formal Analysis of a Process-Oriented Solution* [4] Welch & Pedersen proposed another solution using barriers. In the case of the Reindeer this barrier is a syncing barrier, where all Reindeer need to wait on the barrier until all are there, and can then together wake Santa. For the Elves a more complicated partially syncing barrier is used which waits only for a subgroup of Elves, which are then let through together. This solution neatly solves the Santa Claus problem although is now getting into territory perhaps too complicated for the initial introductory problem Trono first proposed. These barriers are

located as part of the Santa process, meaning that Santa is receiving messages before he is awoken, acting against the requirement that he sleeps until the required number of Reindeer or Elves have arrived. If this architecture was to be kept without Santa doing any work before he should be working then external Stable or vestibule processes would be needed, which could only be used in a non-networked program, and not for our distributed requirements.

In his dissertation Severin Fichtl [5] used a server to act as stable and vestibule to the Reindeer and Elves. This solution was reached due to Bluetooth limitations on the Lego NXT robots he was using to act as individual processes. They could only manage a limited number of connections and so had to each connect to a controller, rather than to each other. This meant that for initial experiments most of the programming was done on this external controller process. Through numerous experiments eventually the PC was reduced to just a hub for communications, though it did still contain various stable and vestibule processes to help manage the Reindeer and Elves. This meant it was easy to count each process in to each stage correctly, but means the solution is far less distributed than Fichtl suggests.

These problems are seen repeated through the literature, such as in the multiple examined solutions in *Solving the Santa Claus Problem, A Comparison of Various Concurrent Programming Techniques*[6]. Some solutions rely on Santa doing the counting in, thereby acting against the first sentence of the original problem, where Santa should only be woken once everything is ready. Alternatively, solutions use external processes to manage the Reindeer and Elves. Most commonly these are stable and vestibule processes, but they could also be barriers or the like. Some of these solutions also end up ignoring the last sentence of the problem, where the last Reindeer should wake Santa. In any case, none of the previously available solutions would be implementable on a distributed system such as over multiple machines without a central controller/hub, or additional external processes.

1.4. Requirements for a Distributed Solution

From the analysis of the problem set out by Trono, and from reading several available solutions we can define the requirements for the solution developed in this paper in Table 1. Most of these requirements come from the original problem, though requirement 4 has been introduced due to the nature of the distributed system. External processes are unfeasible due to there not being a physical space to host those processes, unless they are on the Santa device which would contravene requirement 7. Processes can be made up of sub-processes as this is an effective way of producing re-usable, testable code [1]. Requirements 5 and 6 have been introduced as otherwise the problem would become trivial, with predetermined Reindeer and Elf managers.

2. Setup and Investigations

2.1. Hardware

To implement the solution a network of PCs was used. These PCs are all on the same network as part of a computer lab at Edinburgh Napier University. Each PC is a quad core Intel i5 PC running Windows 10 with a wired internet connection. These machines can run multi-threaded programs, which is necessary as to solve the Santa Problem each process will be broken down into multiple sub-processes. The pre-existing network between the machines will aid communications between them, and reduces the chances of firewalls or other problems interrupting communications. This means that if a problem does occur it is easier to conclude that the problem is with the proposed solution, rather than the physical hardware. As these machines meet all requirements, they are determined to be sufficient for the needs of the project.

Table 1. Requirements for solutions proposed in this report

1	The system should have one Santa process.
2	The system should have nine Reindeer processes.
3	The system should have ten Elf processes.
4	No other processes should be introduced, though individual Reindeer, Elves or Santa may be made up of several processes themselves.
5	All instances of Elves must be running the same code as all other instances of Elves.
6	All instances of Reindeer must be running the same code as all other instances of Reindeer.
7	Santa must be able to sleep and only sleep when not consulting Elves or leading Reindeer.
8	Santa should only be woken once all Reindeer or three Elves are ready to deliver presents/consult.
9	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 till it is over.
10	The Reindeer must go on holiday, each for a random amount of time.
11	When on holiday, each Reindeer should be uncontactable.
12	The last Reindeer to arrive must wake up Santa.
13	The Elves must get to work in the workshop, each for a random amount of time.
14	When building toys, each Elf should be uncontactable.
15	Only 3 Elves at a time should consult with Santa about toys.
16	If there are two groups of Elves looking to talk to Santa, the first to arrive should be served first by Santa.

2.2. Software

To program in parallel, JCSP [7] was selected. This was due to both the authors being already familiar with it. JCSP is a library for Java that provides an implementation of Communicating Sequential Processes (CSP) [8]. CSP is a mathematical definition for the communication between numerous concurrent processes. These processes can be run on the same machine, or separate machines and communicate using synchronus communication. Both the sender and receiver in a synchronus commuciation system need to be ready to communicate for the communication to occur, and if one is ready without the other then it will simply wait for both to be ready.

CSP is a language independent definition, with JCSP being a specific implementation in Java. It enables concurrent and parallel systems to be built within Java, and provides systems for their synchronus communication. As such, it is an appropriate language to solve the Santa Claus Problem, and modified versions of it have already been used in similar projects to this [9].

JCSP can be paired with Groovy, a language which compiles to Java byte code. With some related JCSP libraries it can simplify almost all JCSP programming with greatly reduced verbosity [10]. The primary author has previously only used JCSP with Groovy but it was decided to avoid using it in this project. This was as an eventual aim of the researchers is to have each process be based not on a PC but on robots, and so any programmes potentially will be extremely limited in the digital footprint available to them. As JCSP can function adequately without Groovy, and Groovy is particularly large it was decided not to use it to future-proof the solution.

2.3. Clients and Servers

When constructing a parallel system, as in all systems, care must be taken to ensure that it does not break or run incorrectly. Within a CSP system, or any other message passing parallel system, a client/server model [11] may be used to avoid common pitfalls during the implementation. This is because two common problems in these systems are deadlock and livelock, which the client/server model will address.

A concurrent system with multiple processes talking to each other synchronously in the manner of CSP will have several processes potentially trying to communicate with other processes at any one time. As discussed in the previous section, if these processes are trying to communicate they need to synchronise, with both the reader and writer process ready to communicate. If one process of a pair of potentially communicating processes is not ready, then the other will wait for it to do so. Whilst waiting, a process cannot progress in any other way and so is blocked until the communication can occur. This is acceptable if the waiting is finite, as the process will resume eventually, however it is not always a finite wait. Consider 2 processes, A and B. If A wanted to send a message to B it would wait until B is ready. B is waiting to send a message to A, and so cannot receive from A. Both processes cannot proceed as they are each waiting to be able to send to the other. This will continue forever. This is deadlock and should be avoided as it will stop the system prematurely.

One way to avoid this is to adopt the client/server model [11]. This divides processes into two categories, clients and servers. Clients are processes which can initiate communications by sending messages to other processes. Servers are processes which receive those messages, and if a response is expected, must generate one in a finite amount of time. Finally, if a client expects a response to a message, it must wait immediately after sending its message for the response. This architecture guarantees that responses to messages are properly accommodated without deadlock occurring. If more than two processes are linked together then at least one of them may act as both a client and a server. If all processes progress in a finite time, and there are no server to server communications then livelock can be avoided. This is proved by Welch, Willcock and Justo [11].

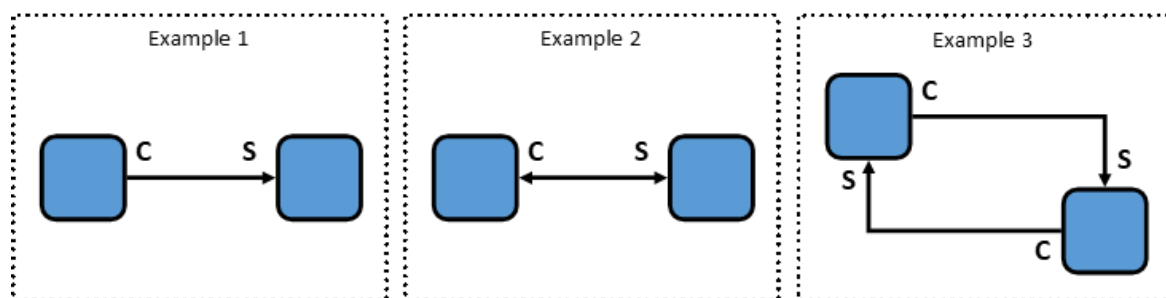


Figure 1. Client/Server examples.

In the examples shown in Figure 1 we have some simple systems. In the first we can see two processes acting as a client and server. This is so simple it cannot deadlock. In the second the server will send a response to the client and so also will not deadlock. The third example is of a situation to be avoided, as the two processes are each trying to act as a client at different points. If they both did this at the same time, then deadlock would occur. This situation can be identified as the connections of processes interactions forms a loop. When we come to design the solution to the Santa Claus Problem, if we can map out the process interactions in such a way we can determine if the system is deadlock free or not. It is worth noting at this point that an absence of loops of client/server interactions guarantees freedom from deadlock, but that loops being present may still result in a deadlock free system if such interactions are properly managed.

Livelock is a similar problem to deadlock in that the system does not progress. However, it is caused by processes being continually busy but not interacting with any external processes, and so the overall system state is never updated. The client/server model will do nothing to prevent this, as it assumes that all processes will proceed in a finite time. We can write our processes to avoid this however, and if we do so then the client/server model will also guarantee that the interactions are livelock free, though this can only be said for certain after rigorous testing, to determine that all processes will always complete in a finite time.

3. Solving the Santa Claus Problem

3.1. Solution Overview

It was decided that the easiest way to solve the Santa Claus Problem would be to break it into two main sections, Reindeer and Elves. The two parts function quite differently and can be treated completely differently as a result. The only point of contact they might have is Santa, who will also need to be implemented. Each Reindeer and Elf would have to be able to talk to Santa, and to all the other Reindeer or Elves as will be explained later on. This gives us the general structure shown in Figure 2.

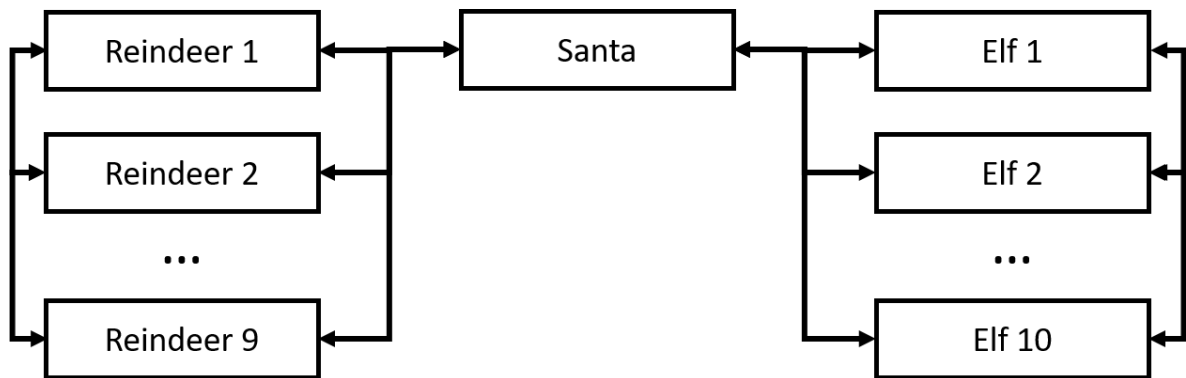


Figure 2. Overall solution architecture. Note that in the implemented solution there are nine Reindeer processes and ten Elves. Only three of each have been shown for brevity.

3.2. Santa

Santa can be a straightforward design as he just sleeps and is able to listen for messages from either the Elves or the Reindeer. As this is all he can do according to requirement 7, there is not much to program within Santa. The process just needs 2 inputs, one from the Elves and one from the Reindeer. These would receive messages from the Elves or Reindeer respectively, informing Santa that a group of Elves or all the Reindeer are ready to consult about toys or deliver presents. As Santa may already be busy the messages are stored in a buffer process before being sent on to the Controller, the Controller being the central process managing Santa's overall interactions and state. The buffer implements a sort whereby any input from the Reindeer is always put at the front of the queue, in accordance with requirement 9. This is all that needs to be done by Santa, with the other requirements on Santa, such as 8 or 12 being implemented within the Reindeer and Elves. This has resulted in a Santa process made up of four sub-processes with the design shown in Figure 3.

In Figure 3 the four processes internal to the Santa process are shown with a light grey background. They are also all delimited by the dotted box. The processes beyond that, shown in the white rectangles are external processes that Santa may interact with. The Elves and Reindeer processes represent every Reindeer and Elf which will connect to the relevant Lis-

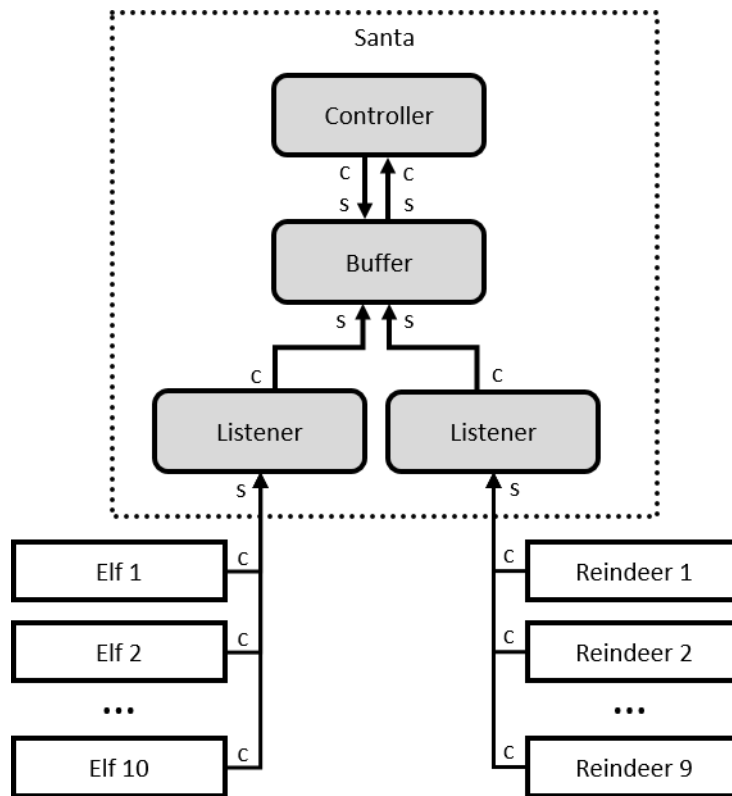


Figure 3. Santa network diagram. Note that in the implemented solution there are nine Reindeer processes and ten Elves.

tener in a many to one connection. Providing the system is implemented as shown, it would be impossible for such a system to deadlock. The network diagram is very simple, with no loops of processes acting as both clients and servers. The only time that messages can be sent by two processes is between the Controller and the Buffer. Note that in both instances the Controller is acting as the client and the Buffer is acting as the server. This is as the Controller is sending a message and will get a response from the Buffer, the buffer never initiates communications. All of this is compliant with the client/server model and so we can say that Santa is guaranteed deadlock free by design.

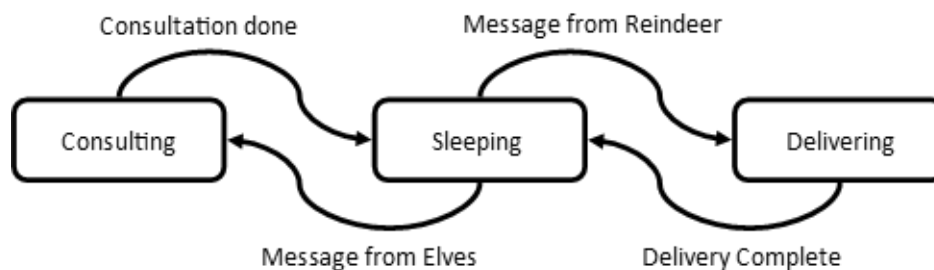


Figure 4. Santa state diagram.

The Santa process implements the state diagram shown in Figure 4, with Santa transitioning between sleeping, and consulting or delivering presents. Note that communication from Santa to the Reindeer or Elves is managed by agent processes sent by the Reindeer or Elves, and so is not covered by the network diagram. At this point we can say that the Santa process is complete as it implements all that is required of it.

3.3. Reindeer

The Reindeer are a significant challenge compared to Santa. They must be able to arrive back from holiday and determine if any other Reindeer have already arrived. To do this we shall use *feeler* messages. As the Reindeer do not know in advance the order of their arrival, when a given Reindeer arrives at the North Pole it will not know which, if any, other Reindeer have also arrived. To get around this, when a Reindeer arrives it will send a message to each other Reindeer to announce that it is now at the North Pole and ready to start grouping up. As each Reindeer doesn't listen when it is on holiday the sent message can only be received once the Reindeer has returned from holiday, and started listening. In other words, when Rudolf is on holiday he does not know in which order all the Reindeer will return to the North Pole, and doesn't know if he will be first, last or somewhere in the middle. Whilst he is on holiday he doesn't listen to any other Reindeer and only tries to interact with them once he gets back. He does this by sending a message to all the other Reindeer to announce that he has arrived. In our example let us suppose that Dasher has already arrived, in which case Dasher will receive this announcement, and so now know that Rudolf is also there. Dasher will also have sent a notification to all Reindeer, which Rudolf will have received upon his arrival, and so Rudolf will know that Dasher has arrived. If Vixen has not arrived then Vixen will receive no messages until their return to the North Pole, despite having been sent a message by both Rudolf and Dasher.

In this way it is possible for a Reindeer to 'feel out' the presence of the other Reindeer through the use of a feeler message. However, if all Reindeer start by sending out a feeler message to the other Reindeer then we will have deadlock, as in JCSP if we start writing a message the code will not progress until the message has been read. This means that if a Reindeer is not there to receive the message as is stated in requirement 11, then the Reindeer that has returned and is trying to contact it will hang until its message has been received. Additionally, as all Reindeer are the same due to requirement 6, we cannot have all Reindeer writing a message as then no one can listen, so the group will deadlock. The same problem is encountered if we get all Reindeer to listen initially. It is possible that timers could be implemented so that Reindeer listen on a timer, and if they receive nothing in a certain time they try writing. Although extremely unlikely this system would be vulnerable to deadlock as different Reindeer will be acting as clients and servers effectively at random.

For this reason, the Reindeer process will be broken down into numerous smaller processes, in a manner similar to Santa. Each Reindeer will have a dedicated Listener process that will constantly listen to input from other Reindeer, and immediately output any messages to a buffer. The Reindeer will also have a dedicated Writer for each other Reindeer. When the Reindeer returns from holiday it determines which other Reindeer are already waiting by sending a message to each of the other Reindeer via the Writers. These writer processes will hang trying to send this feeler message until the Listener in that respective Reindeer is activated. This is because the Listener will only accept messages from other Reindeer when the owning Reindeer is not on holiday. When the holiday has finished the Listener will activate and receive any waiting feeler messages. These feelers will contain the time of arrival of the sending Reindeer, with the times being compared by each Reindeer once they have received all eight. This is then combined with their own time of arrival to get a complete list of nine. The last Reindeer will then contact Santa about delivering presents and they will all be taken off together to do so.

As a quick aside, the time of arrival contained in each feeler message is taken locally within each Reindeer, and so may not be in an absolute chronological order. A global time could be taken, though it could be a rather complicated process in itself to do so. As an alternative it could be that the times are taken not by the Reindeer themselves, but upon their contacting of some central process to count them all in, such as a vestibule or stable.

This also is not a satisfactory solution as the extra vestibule and stable are not suitable for a distributed solution, as previously discussed in Section 1.3. This means that despite the timings potentially being out of sync with each other, localised timings are used within the system. This may be a potential weakness of the system, as it will not necessarily identify the last Reindeer to arrive, but will identify what it *thinks* is the last Reindeer to arrive. Ultimately the difference between the two could be said to be meaningless for the purpose of the Santa Claus problem. As long as a single Reindeer is the one notifying Santa that all Reindeer have arrived then requirement 12 has been met in spirit. This hand waving of requirement 12 may be unsatisfactory however, and future work may address it more completely.

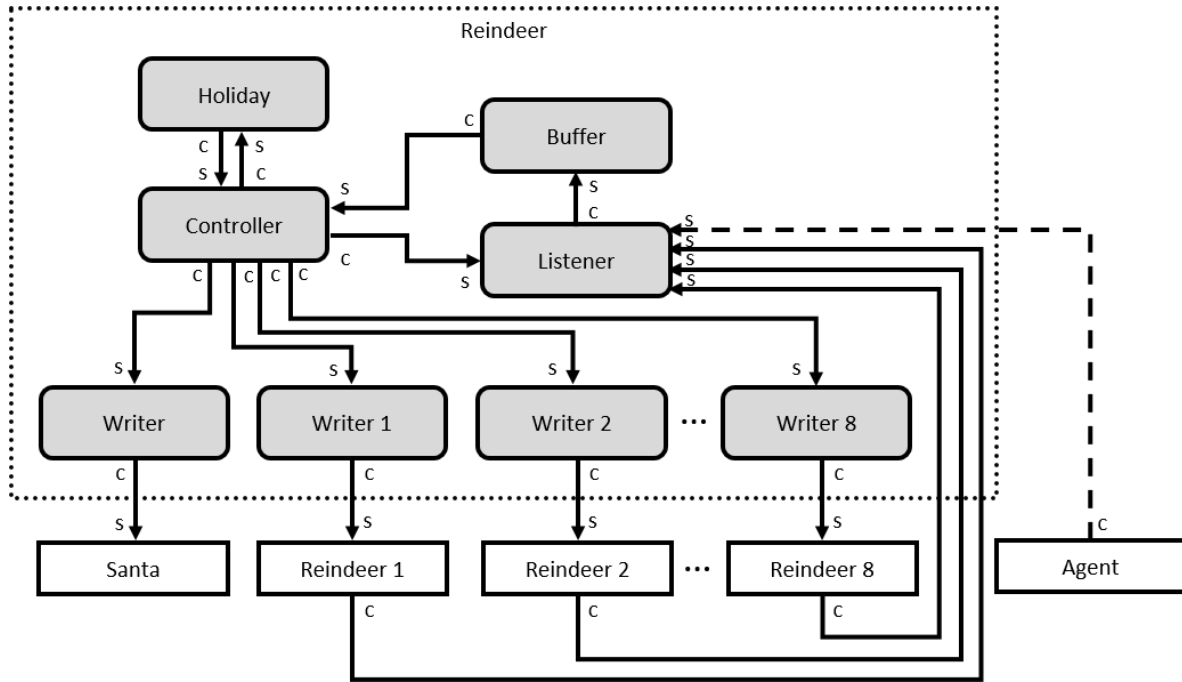


Figure 5. Reindeer network diagram. Note the in the implemented solution there are eight other external Reindeer processes connected to the Reindeer, with eight corresponding Writer processes, each connected to one other Reindeer. All external Reindeer will connect to the same Listener however.

Figure 5 shows the network layout for the Reindeer process. Note that there is one Writer for each other Reindeer in the system. By default, this is eight Reindeer so there are eight Writers each connected to a unique Reindeer. When a Listener receives eight arrival times it sends them as an array to the Controller who will add its own arrival time. This means that all nine Reindeer will have an array of arrival times, each of the same length and containing the same times. The result of this is that when each Reindeer needs to make the decision of which Reindeer arrived last by comparing the times, all come to the same conclusion as all have the same data. This is why the distinction between *actual last to arrive*, and *appears to be the last to arrive* discussed in the last paragraph does not matter. It only matters that all nine Reindeer are making the same decision with the same information, which is the case within this system. Note the dotted line showing communication from the Agent to the Listener. This is because this communication channel is dynamically created at runtime if the Controller is contained within the last Reindeer to arrive, and so does not always exist.

By splitting the Reindeer into dedicated Writer and Listener processes it means that when a Reindeer arrives it can both read and write without deadlocking. This allows all Reindeer processes to be identical per requirement 6, and ensures all Reindeer can find all other Reindeer. Note the interactions between the Controller, Listener and Buffer may potentially breach the client/server model and so lead to deadlock. However, the Controller only sends a message to the Listener when the holiday is finished, the Listener sends all incoming mes-

sages to the Buffer, and the Buffer only writes when a complete array of eight received feelers is received. Deadlock cannot occur in this case as none of these events can ever overlap.

Additionally, there is a loop between the Holiday and Controller processes, where each process acts as a client and initiates communication. Again, these cannot occur at the same time as a Holiday is a simple timer process, told to start by the Controller and then reporting its completion to the Controller. The controller will never start a new holiday without having gone through a whole turn of starting a holiday, finishing a holiday, meeting other Reindeer and delivering presents. This process cannot start without the Holiday process sending its completion message to the Controller. This means that the two messages can never be sent at the same time, and so deadlock cannot occur.

All this means we can still conclude that the system is deadlock free as the two potential trouble spots are known to not cause deadlock. Note the system does require all Reindeer to have prior connections to all other Reindeer between the Writers and Listeners. Future work may remove this limitation.

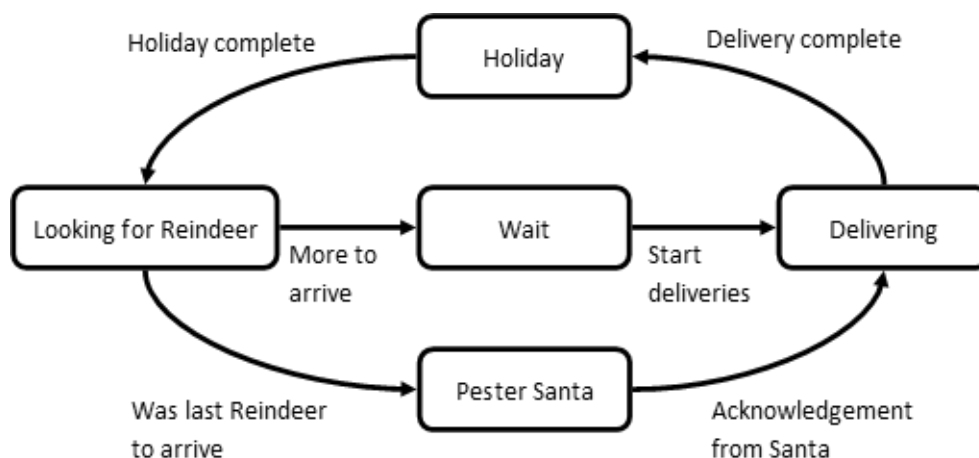


Figure 6. Reindeer state diagram.

3.4. Elves

The Elves will have a similar internal structure to the Reindeer though will interact very differently. Where the Reindeer can wait for all Reindeer, the Elves must only wait for a non-predetermined subgroup of Elves. To model this, we will adopt the idea of a chain. When Elves start looking for other Elves to consult with, they will only consider feeler from other Elves that have arrived later than them. Meanwhile any Elf will be able to be recruited by an Elf that arrived earlier. Each Elf can only have two links, one to an earlier Elf and one to a later Elf. This means that we can think of the Elves as a chain of machines, with the top Elf having arrived first and a chronological ordering down along the chain until the latest Elf is at the bottom. If each Elf can create messages then we need to be extremely careful about when we do so, so as not to deadlock the system. To do this we shall establish the rule that Elves may only send messages up the chain, and must respond down the chain to any messages. No new messages can be sent up the chain whilst an Elf is waiting for the response to a message. Following this simple rule means the system cannot deadlock as it creates an alternating chain of clients and servers, with no client to client communications. This also means we can cut out the inter Elf Writers and Listeners, and have direct communication from Controller to Controller. This reduces the number of processes a message needs to pass through, and so helps reduce the error rate in a system due to various parts having seen (or not) a certain message.

This system relies on their only ever being two links either side of the Elf, and only one other Elf ever thinking that it is connected to an Elf. In other words, two Elves cannot both think that they are front linked, or back linked to the same Elf. This is achieved by an Elf only attempting to front link to one other Elf at a time, with subsequent attempts conditional on the previous one having failed. No attempts will be made to back link to other Elves, with this only happening when one Elf is trying to front link to an Elf. Because the messages are only sent by later Elves to Elves that have arrived earlier no deadlock will occur, as no Elves will try initiate communications with each other, as one will always be later. In the unlikely case of arriving at the same time, the go alphabetically. This system is deadlock free and will allow for a linked chain of all Elves.

However, it is not that simple. There will be occasions where we need to send a message down the chain. This could occur if a chain of four links forms, and so the third Elf needs to send a message down the chain to the fourth letting it go. This will have to be handled by two new processes, the OutOfSyncBuffer and the OutOfSyncWriter. The OutOfSyncWriter process connects directly to Controller processes within other Elves, but has an associated OutOfSyncBuffer so commands can be saved up. This means that messages sent down the chain without an immediate preceding message can be sent to this buffered writer, to only be sent on if the target Elf is free, and ensuring the sending Elf is free to receive any messages itself. This ensures freedom from deadlock. The ensuing network diagram of an individual Elf is shown in Figure 7.

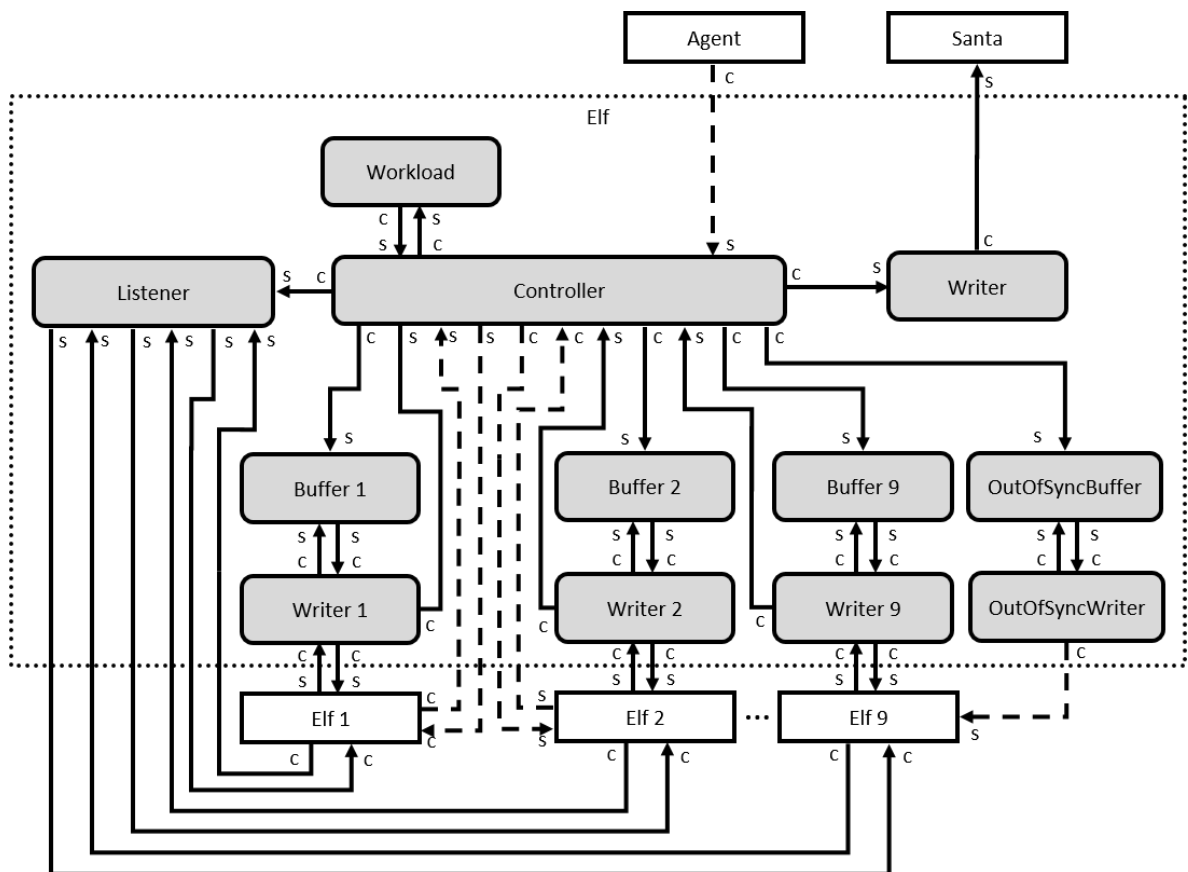


Figure 7. Elf network diagram. Note that in the implemented solution there are nine other external Elf processes connected to the Elf, with nine corresponding Writer processes, each connected to one other Elf. All external Elves will connect to the same Listener however. There is only one OutOfSyncWriter.

The initial discovery of the other Elves is made by the Writer process sending out feeler messages, as in the case of the Reindeer. These are responded to directly by the Listener process with an attached boolean denoting if the feeling Elf is acceptable, e.g. arrived later

than the listening Elf. If the feeler was acceptable or not, the feeler is responded to and the writer sends the response to its controller. In this way all other Elves will be discovered, but to make this possible the Listener must always be listening, even if the Elf is supposed to be working. This conflicts with requirement 14 and further adjustments could possibly be made to avoid this, but it is not a massive sacrifice for a working system.

In the network diagram shown in Figure 7, three types of inter Elf communication are demonstrated to and from the three Elves along the bottom of the diagram. The left most one demonstrates a back linked Elf which will send messages as a client to the Controller, and receive replies as a client. The middle one demonstrates a front linked Elf who will receive communications from the Controller, with the Controller acting as a client and receiving replies as such. Finally, the right most Elf is neither front nor back linked to the Controller, and so any communications would be sent using the OutOfSyncWriter, even if such an Elf were to send messages to the Listener. Note that any Elf may be sent messages by the OutOfSyncWriter as its channels are changed dynamically at runtime, hence the dotted line showing its communications. The two left Elves also have dotted lines between them and the Controller. This is because once a link has been formed direct communication from Controller to Controller can take place, providing it is going up the chain.

There is only one point of concern within this architecture regarding the client/server model, which is the Workload and Controller process. This is functionally identical to the Holiday and Controller interactions within the Reindeer. For the same reason that the Holiday is not a problem, the Workload has also been managed so as not to be a problem. Any other points that may have formed an issue are taken care of by the OutOfSyncBuffer and OutOfSyncWriter processes.

The state diagram for an Elf is shown in Figure 8. Despite it seemingly being far more complicated, it is essentially the same as the Reindeer diagram, with Elves working, looking for other Elves, and contacting Santa. The additional states are used to confirm that a chain has been formed, and to restart looking if an attempted connection did not successfully complete.

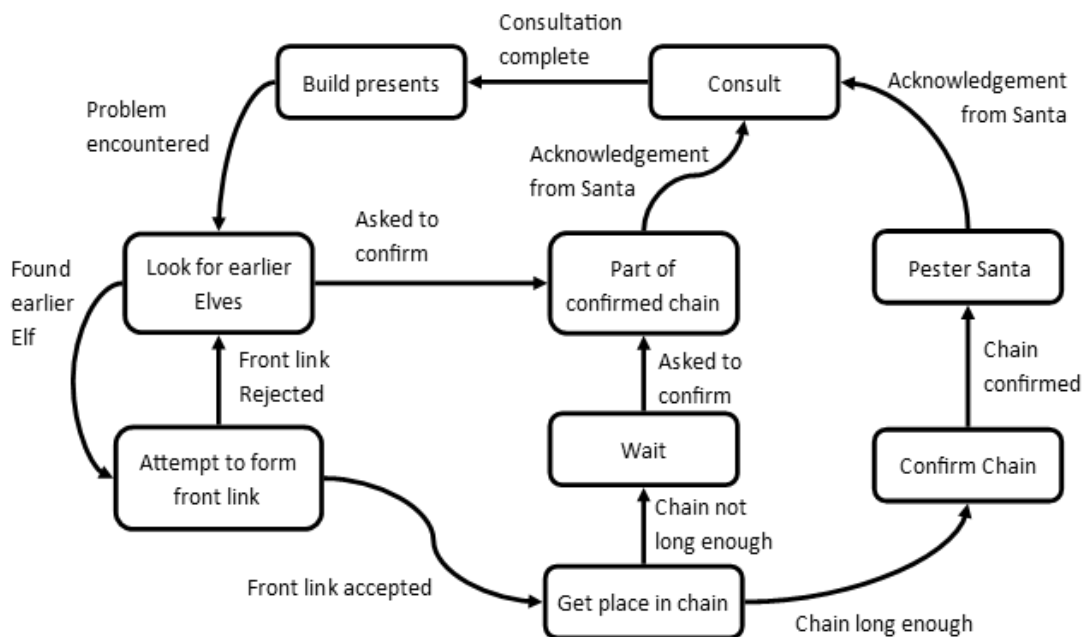
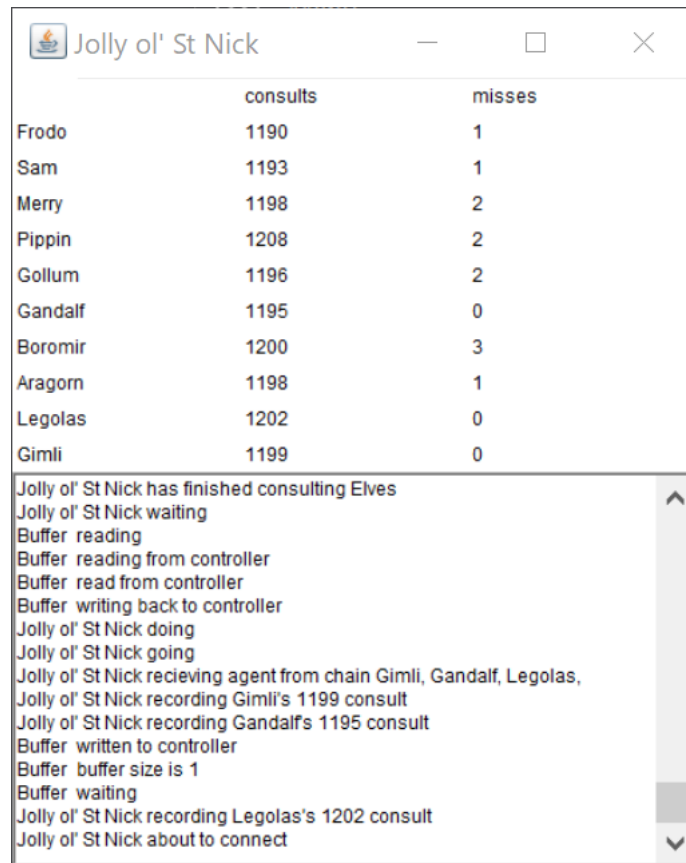


Figure 8. Elf state diagram.

4. Testing the Solution

To test the viability of the proposed solution it was developed within the Eclipse IDE for Java, using JCSP. The solution was run on twenty machines with one each for Santa, the nine Reindeer and ten Elves. Santa prints messages to the Eclipse Console to show when it was either consulting Elves, or delivering presents with Reindeer. This was sufficient for the Reindeer who must occur as a group repeatedly. The Elves however required the development of a specially built console window. This window is tied to the Santa process and lists Santa's interactions with other processes, a list all the Elves within the system, a counter of how many times each Elf had consulted with Santa, and how many consultations have happened since the last time that Elf had been involved in a consultation. This was necessary as it allows us to check that individual Elves are not deadlocking or livelocking, with the system continuing without the wider system noticing. As long as the number of misses remains small we can be certain that the given Elf has not deadlocked or livelocked. A sample output of this system is shown in Figure 9.



	consults	misses
Frodo	1190	1
Sam	1193	1
Merry	1198	2
Pippin	1208	2
Gollum	1196	2
Gandalf	1195	0
Boromir	1200	3
Aragorn	1198	1
Legolas	1202	0
Gimli	1199	0

```

Jolly ol' St Nick has finished consulting Elves
Jolly ol' St Nick waiting
Buffer reading
Buffer reading from controller
Buffer read from controller
Buffer writing back to controller
Jolly ol' St Nick doing
Jolly ol' St Nick going
Jolly ol' St Nick recieving agent from chain Gimli, Gandalf, Legolas,
Jolly ol' St Nick recording Gimli's 1199 consult
Jolly ol' St Nick recording Gandalf's 1195 consult
Buffer written to controller
Buffer buffer size is 1
Buffer waiting
Jolly ol' St Nick recording Legolas's 1202 consult
Jolly ol' St Nick about to connect

```

Figure 9. Custom console Output. It has been noted that the names used are not the names of elves, but the author was not aware of any recognisable groups of 10 elves to use.

The proposed solution implements all the requirements set out in Section 1.4 apart from requirement 14, that all Elves should be uncontactable when building toys. This is as the listener process of each Elf always needs to be active to prevent deadlock. This work around was determined to be easier to implement to accommodate the sheer unpredictability of the system, with Elves coming and going effectively at random, and messages between Elves being unpredictable in their timing. With further work this requirement may be kept to. It is also questionable as to if requirement 12 that the last Reindeer wakes Santa has been met, due to the use of non-global timers. As has been argued in Section 3.3 this should not affect

the solution in anything other than an academic sense. It may however be a cause for further work to do away with this handwave.

4.1. Results

The complete system with Santa, nine Reindeer and ten Elves was run for over 24 hours without any processes deadlocking, livelocking or encountering a detectable error. Whilst we could have said that the system is deadlock and livelock free ahead of time from the client/server modelling that has been done, the test is still informative as it shows that it is free from processes taking infinitely long, and that the system has been implemented as modelled. Within the timeframe available to this project this is the best manner of testing available as there is insufficient time to carry out a complete formal analysis, which is anecdotally expected to take many months of work and be a project in its own right. The solution has been tested on a network of machines and so can be considered a distributed system.

4.2. Client/Server

Although a complete formal analysis of this system cannot be done in the time available, the architecture can be examined with relation to the client/server model as a way of showing that it is guaranteed to be deadlock and livelock free. Santa, the Reindeer and the Elves have all been demonstrated how they are deadlock and livelock free in Sections 3.2, 3.3 and 3.4 respectively. All systems together can be easily demonstrated as deadlock free as each one can be expressed as a single process as shown in Figure 10 below.

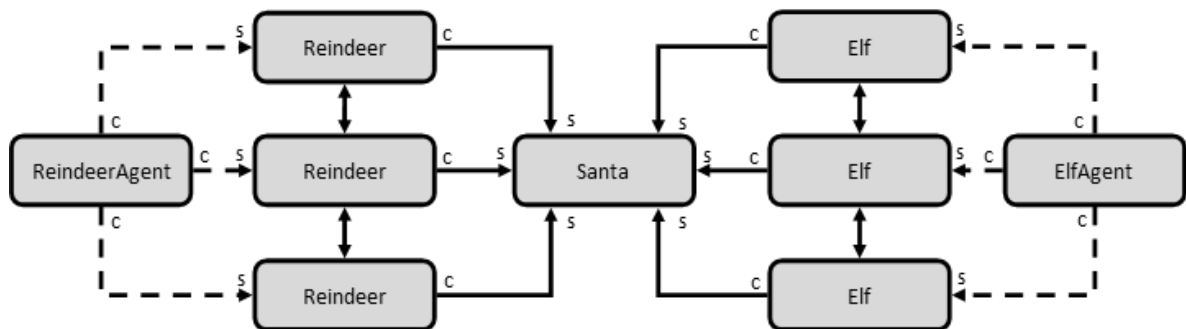


Figure 10. Network diagram of the entire system including agents. Note that all Reindeer are connected to all other Reindeer, and all Elves are connected to all other Elves. In the complete system there are nine Reindeer and ten Elves but only three of each have been shown for brevity.

From Figure 10 we can see that no deadlock can occur between the processes as there are no client/server loops. Note that the interactions between the Reindeer and between the Elves have not been shown in any depth as at this stage they are irrelevant. Communications only occur at this level, between Reindeer and Santa or between Elves and Santa if communications amongst the Reindeer or Elves has completed. Also note there are no communications between Santa and either of the Agents, as the Agents function more as classes rather than separate processes that are run. For this reason, it is definitionally impossible for them to deadlock. As we have demonstrated now that no part of the system can deadlock or livelock according to the client/server model, we can conclude that the system is complete and sufficient for our requirements.

5. Conclusion

This paper set out to demonstrate a working solution to the Santa Claus Problem that could work on a distributed system, such as between multiple desktop PCs. The requirements of

such a system are defined, a system is described and finally it is implemented and tested. The system breaks down each of Santa, Reindeer and Elves into several sub-processes, enabling them to attempt to both read and write at the same time without deadlocking. This enables the processes to dynamically discover each other and sort themselves into the required sets to contact Santa according to the problem definition. Of particular note is the development of the Elf process as it uses a novel chain approach, allowing the processes to discover each other without deadlocking the system. The system has been tested sufficiently to state with reasonable certainty that no errors exist within it, and the architecture is deadlock and livelock free. Furthermore, it has been examined using the client/server model to determine that no deadlock or livelock can occur. For this reason we can conclude that the proposed solution is correct and complete.

The question then arises whether this still satisfies Trono's original requirement to provide an introductory problem for students to better understand concurrent and parallel systems. Simple solutions do exist if the solution uses vestibule and stable processes, barriers or even altering barriers. The solution presented here is definitely one that cannot be considered simple but then it does solve a redefined version of the problem that was not even feasible when Trono first proposed his problem. However as a means of exploring the problems posed by distributed systems it could be used as a means of introducing the additional considerations, especially if the caveats of Trono's original problem are respected.

References

- [1] Jon Kerridge. *Using Concurrency and Parallelism Effectively*. Bookboon, 2014. ISBN: 978-87-403-1038-2.
- [2] J. Trono. A New Exercise in Concurrency. *ACM SIGCSE Bulletin*, 26(3):8–10, 1994.
- [3] M. Ben-Ari. How to Solve the Santa Claus Problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.
- [4] P. H Welch and J. B. Pedersen. Santa Claus: Formal Analysis of a Process-Oriented Solution. *ACM Transactions on Programming Languages and Systems*, 32(4):485–496, 2010.
- [5] J. B. Fichtl. Using a Parallel Solution to the Santa Claus Problem to Investigate the LEGO Robot's Bluetooth Functionality. Unpublished dissertation from Edinburgh Napier School of Computing, 2010.
- [6] J. Hurt and J. B. Pedersen. Solving the Santa Claus Problem: A Comparison of Various Concurrent Programming Techniques. *Communicating Process Architecture*, pages 381–396, 2008.
- [7] CSP for Java(JCSP). CSPforJava@bintray.com.
- [8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.
- [9] J. Kerridge, A. Panayotopoulos, and P. Lismore. JCSPre: The Robot Edition to Control LEGO NXT Robots. *Communicating Process Architecture*, pages 255–270, 2008.
- [10] Apache Groovy. <http://groovy-lang.org>.
- [11] P. Welch, G. Justo, and C. Willcock. High-Level Paradigms for Deadlock Free High Performance Systems. *Transputer Applications and Systems*, pages 981–1004, 1993.