Distributed Algorithms: An Approach to Autonomous Satellite Constellations

Modelling, Developing & Visually Simulating Distributed Algorithms for Autonomous Satellite Constellations

Master's Thesis

SW10

Aalborg University
Software

Summary

Due to the development of miniaturised satellites, current satellite constellation projects, such as StarLink, consist of many more satellites than existing constellations. Additionally, future deep space missions, at e.g. The European Space Agency, will launch formations of satellites, instead of a single large satellite. Due to their miniaturised size, the satellites are more constrained in terms of battery life and computation power, and can be affected by tumbling more easily. For this reason, we look at distributed algorithms for autonomously maintaining a functional constellation or formation, as broken links may severely affect the performance of a network of satellites. More autonomous satellites reduce the need for communication between the satellites and earth, while also reducing the need for manual maintenance. These algorithms should be effective in terms of the number of transmissions and computation, to reduce the energy cost of the satellites.

We look at networks of satellites that communicate in a ring network, making it suitable for satellite constellations and usable for formations of satellites that always have one path that connects all satellites. Here, we specifically look at how satellites should detect and reason about potentially failed satellites and generate a new constellation or formation if needed. We also look at satellites discovering and handling new additions of satellites to the constellation or formation. Our distributed algorithms are modelled and verified separately in UPPAAL. Here, we verify that the algorithms produce a correct result in each satellite. We also verify the communication complexity.

To see and understand how these algorithms work in combination and in a realistic environment, we have created a framework that simulates a satellite constellation and visualises it in the Unity rendering engine. Here, each satellite is in its own thread and the communication between the satellites are asynchronous. The simulation allows for easily adding or disabling satellites or triggering specific behaviour while the satellites are live, and showcases how the algorithms handle dynamic changes of the constellation and satellites.

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfsvej 300
Phone: +45 99 40 99 40
Fax: +45 99 40 97 98
http://www.cs.aau.dk

Title:

Distributed Algorithms: An Approach to Autonomous Satellite Constellations - Modelling, Developing & Visually Simulating Distributed Algorithms for Autonomous Satellite Constellations

Project:

Master's Thesis

Project period:

01/02/2020 - 12/06/2020

Project group:

DS108f20

Authors:

Bastian Bresson
Jens Burla Johansen
Kasper Hangaard Røgen

Supervisors:

René Rydhof Hansen
Kim Guldstrand Larsen

Pagecount: 53

Hand-in date: 04/06/2020

Abstract:

With the development of more capable satellites, satellite constellations and formations also become more capable. Recent constellations and deep space projects are much larger, in terms of the number of satellites, than previous projects. With these larger constellations, the need for autonomous satellites becomes more apparent, as manually managing these constellations would be infeasible. In this project we explore how satellites could become more autonomous.

This project contains three primary contributions. The first contribution is the development of an autonomous self healing satellite constellation framework. The second contribution is the development of a simulating tool, which allows debugging and interaction with the constellation framework in realtime. The last contribution is the development of multiple UPPAAL verification models, as well as the results from said verifications. These are used to prove the correctness of the framework, as well as to analyse the performance of the framework. The framework, simulator and models serve as a proof-of-concept for an autonomous satellite constellation system.

Contents

Preface

This Master's Thesis is written by three software engineering students during the fourth and last semester of the Master in Software Engineering.

We would like to thank our supervisors René Rydhof Hansen and Kim Guldstrand Larsen for their help, feedback, and suggestions during this project.

Aalborg University, June 4, 2020

Bastian Bresson
<bbress15@student.aau.dk>

Jens Burla Johansen
<jbjo15@student.aau.dk>

Kasper Hangaard Røgen
<kragen15@student.aau.dk>

1 Introduction

The need for more autonomous satellites is becoming more apparent with the development of space technology. This is seen in projects like Starlink, but also applies to Deep Space missions, that i.e. the European Space Agency is planning. The Starlink project is a satellite constellation that will provide global internet access. The Starlink project has been approved to launch 12.000 satellites into orbit. This is the largest constellation project, as the current number of active satellites in Earth's orbit is 4000 in comparison.

These projects are possible due to more sophisticated satellite technology and the development of smaller and smaller satellites. Starlink is capable of launching 60 satellites into space with each rocket, due to their form factor. With smaller satellites, future Deep Space missions are planned to piggyback a number of smaller satellites to assist a larger satellite. Additionally, research in miniaturised satellites further increases the feasibility of launching and utilising more as well as smaller satellites compared to one single, large satellite.

The increasing number of satellites in satellite constellations and space missions, is especially where the need for autonomous satellites is seen. Manually managing larger distributed systems like the mentioned projects would be infeasible and in Deep Space missions, the communication time between the Earth and the satellites would make it impossible to react in a timely manner.

In this project we explore satellite constellations where one wants all satellites to be connected as a network at all times. This introduces some challenges that are to be solved in order to maintain such a constellation, namely:

- Detect failed links
- Reroute messages
- Reorganise the constellation
- Map constellation topology
- Detect & introduce new satellites

In our previous work [1] we investigated three cases regarding failed nodes in satellite constellations: single failed node, double sequential failed nodes, and double split failed nodes. We found that the double split failed nodes case has the largest performance degradation in regards to the number of messages that was successfully sent and received in the constellation. Therefore, reorganising the constellation in order to deal with the double split failed nodes case is especially interesting to address.

Modelling & Verification 2

In this chapter we introduce some of the UPPAAL branches as well as UPPAAL's and UPPAAL SMC's modelling and query languages.

Modelling and verification concerns itself with modelling realtime systems and verifying properties about them. Other tools for modelling and verification may be better for our work, however, we stick with UPPAAL as we have experience with it. By sticking with UPPAAL the whole modelling and verification process is faster and the results are of higher quality than if we use another tool. We use UPPAAL to model autonomous satellites in constellations and verify properties about them. Multiple branches of UPPAAL have emerged that focus on different tasks. In the list below, a number of these UPPAAL branches are briefly described. In this work we only make use of UPPAAL and UPPAAL SMC.

- UPPAAL SMC is used to perform statistical model checking, which monitors some properties for multiple runs of a model [2]. Statistics of the runs are then used to determine to which degree the properties hold.
- UPPAAL CORA performs cost optimal reachability analysis using Linearly Priced Timed Automata (LPTA): An extension of timed automata [3]. LPTA allow for annotation of costs which UPPAAL CORA uses to calculate cost-optimal paths that satisfy some goal condition(s).
- UPPAAL TIGA is used to solve timed games with respect to reachability and safety properties based on Timed Game Automata (TGA) [4]. An efficient on-the-fly algorithm is used to solve these timed games.
- UPPAAL Stratego allows for generation, optimisation, and comparison of strategies for stochastic priced timed games [5].

2.1 UPPAAL

UPPAAL is a modelling and verification toolbox designed for realtime systems that can be modelled as networks of timed automata extended with integer variables, structured data types and channel synchronisation [6].

A timed automaton is a finite-state machine extended with clock variables that evaluate to real numbers and progress synchronously. Systems modelled in UPPAAL consist of a network of these timed automata running in parallel and extended with bounded discrete variables. The state of such a system is made up of the locations of all the timed automata, the clock constraints, as well as the values of the discrete variables. Separately or whilst

synchronising with another automaton, an automaton may fire an edge which leads to a new state.

An autonomous satellite constellation should be able to coordinate amongst them which satellites take which locations in a new constellation that might be requested. Figure 2.1 includes a timed automaton modelling a satellite that that runs a simple version of a constellation-plan generation process that handles this coordination.

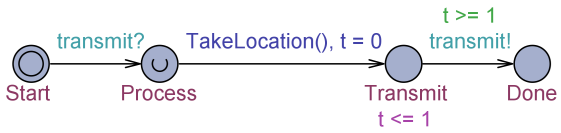


Figure 2.1: Simple plan generation model

All satellites begin in the *Start* location of the plan generation process. Either a satellite or a ground station on Earth may communicate to a satellite to actually begin this process by synchronising on the *transmit* channel. Once the *process* has started on a satellite it *urgently* has to *take a location* in the new satellite constellation while also resetting the clock *t*. The clock t is used to make sure *transmit* only takes *1* time unit by the use of an *invariant* and a *guard*. The transmission of the plan to the next satellite is done using the *transmit* channel. When a satellite has taken a location in the plan of the new constellation and forwarded it to the next satellite, it is *done*.

In UPPAAL, timed automata are extended with the following features via the modelling language [6]:

- *Templates* of automata are declared with a list of parameters of any type which are substituted for an actual parameter in process declaration.
- *Constants* can be defined as *const name value*, cannot be modified, and must be an integer.
- *Bounded integer variables* are defined as *int[min, max] name* where *min* and *max* are lower and upper bound, respectively. The default range is -32768 to 32768, however, if a smaller range is defined the verification time is decreased.
- *Binary synchronisation* channels can be defined as *chan name*. An edge with *name!* synchronises with *name?*. If multiple pairs of channel synchronisations are available, one is chosen non-deterministically.
- *Broadcast channels* are defined as *broadcast chan name*. A sender *name!* is able to synchronise with arbitrarily many receivers *name?*. If a receiver is able to synchronise, then it must. In case no receivers are available, the sender *name!* can still carry out the action. This is to say that sending is never blocked with broadcast channels.
- *Urgent locations* do not allow time to pass when a system is in such a location.
- *Committed locations* restrict the execution even more than urgent locations. If any location is committed, the state is committed. No delay may happen in a committed state and next transition must involve an outgoing edge of a committed location.

- *Arrays* can be declared as *type name[size]* where *type* is the type of the data in the array, *name* is the name of the array variable, and *size* is the number of elements in the array.
- *Initialisers* are for initialising integer variables or arrays: *int i := 3;* or *int i[3] := 3, 3, 3;*.

Expressions in UPPAAL include clocks as well as integer variables and are named by the following labels.

- Guard is an expression that must evaluate to true in order for the edge to be enabled. It has no side-effects, evaluates to a boolean value. References are only to clocks, integer variables, and constants (or arrays of these types). Clocks and clock differences must only be compared to integers. Guards including clocks are conjunctions (integer conditions allow disjunctions).
- Synchronisation is essentially communication between processes where they synchronise on edges. It is on the form *Expression!*, *Expression?*, or empty. The expression must have no side-effects, evaluate to a channel, as well as only refer integers, constants, and channels.
- Assignment is a list of expressions that are evaluated, possibly assigning the evaluations to variables. It is a list of expressions and may have side-effects. Expressions may only refer to clocks, integer variables, and constants. Only assign integer values to clocks.
- Invariant is an expression that must evaluate to true as long as the system is in the location that holds it. It has no side-effects. Only references clocks, integer variables, and constants. Conjunction of conditions on the form $c < e$ or $c <= e$ where c is a clock reference and e evaluates to an integer.

UPPAAL's query language includes state and path formulae. State formulae express individual states while path formulae reason about paths of a model. Path formulae come in three different classes: *reachability*, *safety*, and *liveness*. However, reachability properties are often actually safety properties as they often are used to check if a bad state can be reached. So in reality, there are only two major groups of properties: safety and liveness. Safety properties are concerned with whether a system can ever reach a bad state, whereas liveness properties are concerned with whether a good state can always be reached and thus guarantees progression in the system. In the list below, state formulae and the different classes of path formulae are further described.

- *State formulae* are expressions that can be evaluated for a state. Their syntax is a superset of that of guards, but disjunctions are allowed. One can also test if some process P is in a particular location l using the form $P.l$. The special state formula *deadlock* allows one to check if a system is in a deadlock state; no outgoing action transitions from the state or any of its delay successors.
- *Reachability properties* assess whether or not there exists a path starting from the initial state where a state formula ϕ is satisfied somewhere along the path. In UPPAAL this is formulated as $E<> \phi$.

- *Safety properties* are expressed positively in UPPAAL and come in two forms. In both forms ϕ is some state formula. The first one asks if ϕ is satisfied at all times along all possible paths. This is written as *A[]* ϕ in UPPAAL. The second form expresses whether a maximal path exists so that ϕ is always true and is expressed as *E[]* ϕ in UPPAAL. A maximal path is a path that is either infinite or where the last state has no outgoing transitions.
- *Liveness properties* are on the form "something will eventually happen" and also come in two forms. In both forms ϕ is some state formula and in the second form ψ is also some but different state formula. The first form assesses whether ϕ is eventually satisfied expressed as *A<>* ϕ in UPPAAL. The second one expresses whenever ϕ is satisfied, eventually ψ is also satisfied. In UPPAAL this is written as ϕ *–>* ψ.

A visualisation of the five kinds of path formulae can be seen in Figure 2.2.

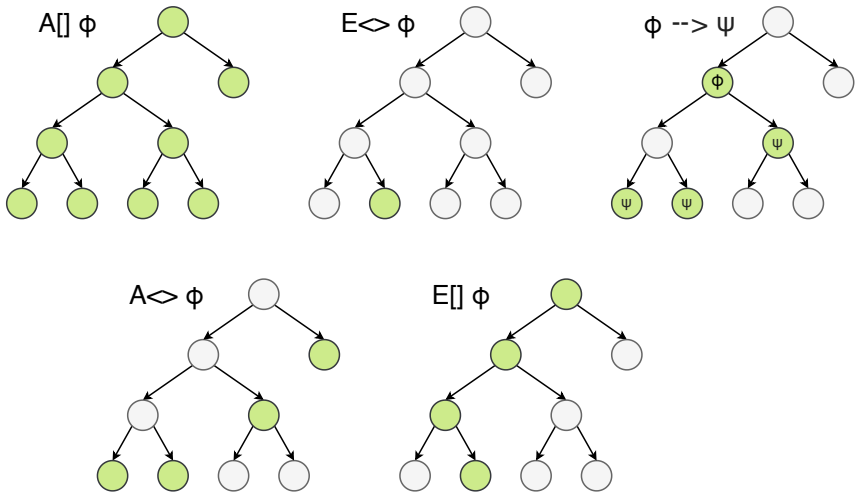


Figure 2.2: Path formulae

2.2 UPPAAL SMC

UPPAAL SMC is a branch of UPPAAL that performs statistical model checking by monitoring some properties over multiple runs and performing statistical analysis to reason about whether they hold or not with some confidence [2].

UPPAAL SMC extends the modelling language of UPPAAL with a stochastic interpretation. For individual timed automata components, non-deterministic choices between multiple transitions are refined by probabilistic choices. Similarly, non deterministic choices for time delays are refined by probability distributions. At the component level these are either given by uniform distributions in cases of time-bounded delays or exponential distributions (with user-defined rates) in cases of unbounded delays.

A simple UPPAAL model including the extensions mentioned above can be seen in Figure 2.3.

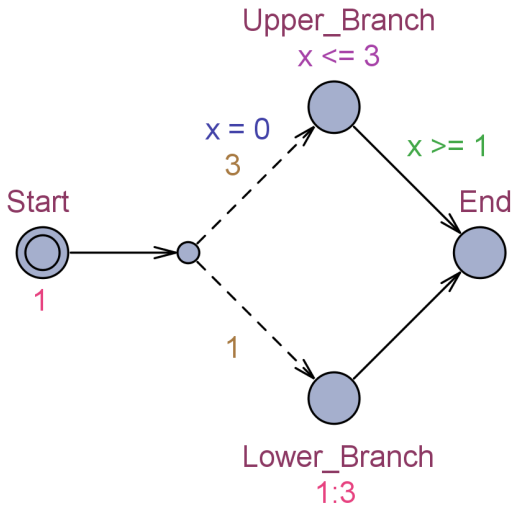


Figure 2.3: Simple UPPAAL SMC model

Ignoring the UPPAAL SMC extensions for a second, one can see that the *End* location can be reached within the time-interval $[0, +\inf[$ due to unbounded time delays in *Start* as well as *Lower Branch*. The stochastic interpretation that UPPAAL SMC provides, refines this interval in terms of a distribution of the reachability time. The unbounded delays in the *Start* location are chosen from an exponential distribution with the rate *1*. The *Start* location leads to a branching point with probability weights *3* and *1*. These probabilities affect the distribution by taking the upper branch 3/4 times and the lower branch 1/4 times when performing runs of the model. The time-bounded delays caused by the invariant $x <= 3$ and the guard $x >= 1$ are uniformly distributed, though, combined with the weight of 3/4 from the branching point. The unbounded delays in the lower branch are chosen from an exponential distribution with the rate *1:3 (1/3)*. Again, this distribution is of course affected by the weight 1/4 from the probabilistic choice at the branching point. The resulting reachability time distribution can be seen in Figure 2.4.

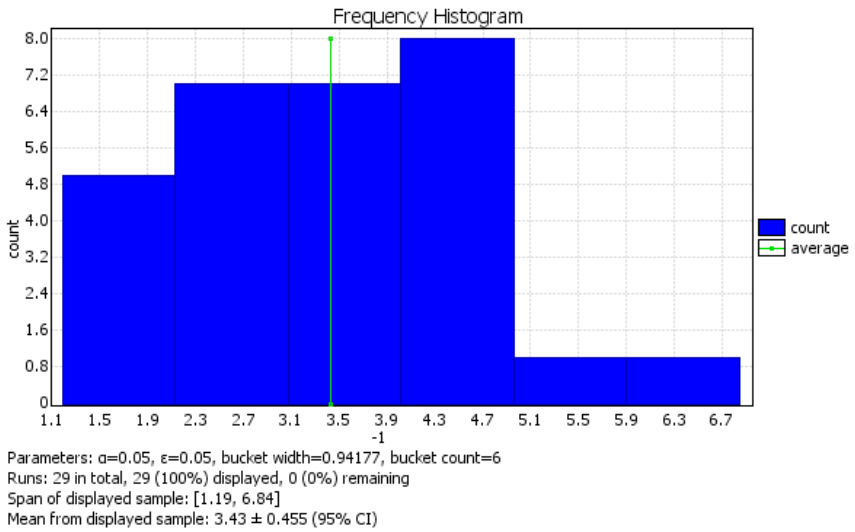


Figure 2.4: Reachability time distribution

UPPAAL SMC also extends the query language by introducing simulation and statistical model checking. Simulation allows for visualisation of expressions throughout runs giving valuable insight about the behaviour of the model. Statistical model checking enables the user to ask more interesting properties. Simulation and some of the different statistical model checking properties are further described in the list below.

- *Simulation* allows for visualisation of expressions throughout runs giving valuable insight about the behaviour of the model. This is done in UPPAAL SMC using the query *simulate N [<=bound] {E1, ..., Ek}*. *N* is the number of runs to be performed, *bound* is the time bound on the runs, and *E1, ..., Ek* are the expressions that are to be visualised.
- *Probability estimation* computes an approximation interval $[p-\epsilon, p+\epsilon]$ for $p = Pr(\phi)$ with a confidence $1-\alpha$. ϵ and α are user-defined. This is expressed as *Pr[bound] (ϕ)* in UPPAAL SMC. *Bound* is the time bound and ϕ is the property that is checked.
- *Probability comparison* allows for comparing probabilities of two different properties. It is defined as *Pr[bound1] (ϕ1) >= Pr[bound2] (ϕ2)*. *Bound1* and *Bound2* are the time bounds while *ϕ1* and *ϕ2* are the properties that are checked.
- UPPAAL SMC also supports computation of expected values of min or max of an expression. This is defined as *E[bound; N] (min: expr)* or *E[bound; N] (max: expr)*. *Bound* is the time bound, *N* is the number of runs, and *expr* is the expression to evaluate. A confidence interval is also given for these properties.

Satellite Constellations 3

In this chapter, we explain the basis of satellite constellations in our framework and introduce assumptions about, and motivation for autonomy in constellations. Furthermore we discuss working with distributed systems, introducing challenges and properties. Lastly we introduce and relate our work to similar work.

3.1 Latency-Critical Constellations

This project is based on the idea of latency-critical constellations. By this, we mean a constellation where the latency of message propagation is of utmost importance. The latency of messages in constellations can be compromised in different ways, one of the primary being a lost connection, either due to a dead satellite, or due to a satellite being temporarily unavailable. Losing satellite links can mean that messages have to be rerouted through the entire constellation, or that a message can never reach its destination. In space, messages are often lost, and as such, reducing the amount of hops in the communication route can greatly reduce the risk of messages being lost, as well as the transmission time of the message. In this project, lost connections will be resolved autonomously by the constellation itself, in an attempt to reduce the downtime of satellite links.

3.2 Autonomy

Autonomy in satellite constellations is an increasingly sought after property. Traditionally, satellites have been using hard-coded contact plans, created by the ground stations in order to function. This makes the satellites dependent on people on the ground. Furthermore, dependency on the ground makes the satellites' ability to react to challenges delayed, as they will have to wait for instructions. If satellites can be created with an increased level of autonomy, they can be able to handle some of the challenges on their own, decreasing the reaction time as well as reducing the need for bandwidth to and from the earth. In this project, autonomy has been implemented as a series of distributed algorithms, with most of them utilising consensus in the network. The communication is based on token-ring, where a token is passed around the network, allowing one node to speak at a time per operation. Every node will have a chance to either increase it's own knowledge, enlighten the rest of the network, or entirely change the currently executing operation, as the node sees fit. The distributed algorithms in this project heavily leverages local knowledge of the nodes, and gives them the power to make decisions without the approval of the rest of the constellation, in an effort to save time and messages.

3.3 Distributed Systems

Distributed systems consist of multiple computation entities with their own local memory. These computation entities then communicate with each other via message passing. The computation entities can be on the same physical machine but is usually on separate machines connected in a network. Satellite constellations are distributed systems with each satellite having a computation unit and a radio module to communicate with other satellites. Some common properties about distributed systems are summarised in the list below.

1. A distributed system may have a common goal that each node contributes to solving. The distributed system is seen as one unit and coordination of which node does what is handled internally.
2. The system must be fault tolerant in case of failures in individual nodes. The nodes may exhibit different unwanted behaviours like not communicating, communicating wrong or malicious messages, or communicating too much (essentially denying service of a node).
3. The structure (e.g. topology and latency) of the distributed system may not be known in advance or may change during execution. This means that the system continuously has to be aware of the structure and changes to it as well as actively try to keep its knowledge of it up-to-date.
4. Each node may have incomplete or out-of-date knowledge about the structure as well as input and state of the collaboratively executing program. Complete synchronisations of knowledge may be attempted, however, the success of this depends on how often changes happen and how long time it takes to complete the synchronisation.

Based on the goals set for this project in Chapter 1, properties 1, 3, and 4 from the list above are apparent or are the ones we need to deal with. Property 1 is apparent regarding the *Reorganise constellation* goal as the constellation should be able to reorganise itself by coordination between the satellites. Property 2 is not dealt with in this work, which is discussed in subsection 4.1.2. The 3rd and 4th properties are apparent in regards to the *Detect failed links*, *Map constellation topology*, and *Detect & introduce new satellites* goals. The structure of the constellation is changing due to both new and failing satellites as well as reorganisation of the constellation. The satellites should have an up-to-date view of the structure in order to pass messages effectively and efficiently. Effectively meaning that a satellite should not try to message another satellite that it cannot reach. Efficiently meaning that a message should be sent via as few hops as possible.

3.4 Related Work

In [7] they propose several distributed algorithms that solve optimisation problems by propagating messages around a ring network. Each node helps solving the optimisation problem with the knowledge it has. In some cases one cycle through the network is enough, however, in other cases multiple cycles are needed to converge at the solution to the optimisation problem. This relates to our approach to solving reorganisation of a constellation. In our approach we solve a fuel optimisation problem where each satellite takes a location and forwards the problem to one of its neighbours. Our approach differs by always solving the problem in one cycle as well as using a second cycle to propagate an execution command to execute the plan.

In [8] they present a novel solution to autonomous task allocation in self-organising satellite constellations. Their method allows satellites to cluster above high-demand areas on the Earth's surface. The solution's efficacy is demonstrated via three different setups of satellites and tasks. They simulated a large constellation of 100 satellites to demonstrate the use of their method for large-scale constellations. Their approach yields efficient solutions and is robust to satellite failures. This relates to our work on many aspects. Like our work, theirs includes autonomy in satellite constellations, optimisation of solutions, and robustness to satellite failures. Our work differs in that autonomy, optimisation, and failure handling is centred around maintaining a fully connected constellation at all times.

In [9] they propose an algorithm that performs automatic topology discovery without complete knowledge. Their approach is able to derive topology of Ethernet networks with up to 2000 nodes and 50 bridges. Topology information is gathered by the bridges whilst forwarding packets. The gathered information is then extracted to get the full knowledge of the network topology. This relates to our method for mapping constellation topology as well as detecting and introducing new satellites. In our method we also gather information about changes in the network and propagate those changes forward. Our method differs by detecting network changes at individual nodes and propagating them around the constellation instead of collecting them at junctions.

Related work to ours has been carried out that look into autonomy, communication, optimisation, failure handling, and topology discovery. However, to the best of our knowledge, no work has been carried out that resulted in the algorithms, visualisation and verification that has been produced in this work.

Distributed Algorithms 4

In this chapter we introduce our framework and our distributed algorithms for generating constellation plans, network discovery, routing, and fault handling. We also discuss theory related to distributed algorithms regarding synchronous and asynchronous environments as well as stopping and byzantine failures

4.1 Theory

Here, we present two important topics regarding distributed algorithms, synchronous and asynchronous algorithms, and handling stopping and byzantine failures. We use the work of Nancy A. Lynch, *Distributed Algorithms* [10], in which these topics are described alongside several algorithms and proofs.

4.1.1 Synchronous and Asynchronous Algorithms

In *Distributed Algorithms*, [10], distributed algorithms and their behaviour are separated in two major groups, namely: synchronous and asynchronous. In synchronous algorithms, they use the notion of *rounds*, in which nodes produce the next message, sends the message, and handles incoming messages. All nodes perform their round at the same time, hence it being synchronous. Due to synchronous nodes computing and sending messages at exactly the expected time, synchronous distributed algorithms are less complex than distributed asynchronous algorithms. Additionally, synchronous algorithms can utilise certain synchronous properties. As an example, the *diameter* of the graph can be used in some algorithms, as the number of rounds nodes should take before terminating. Here the *diameter* is the longest distance between any pair of nodes in the graph, and guarantees, in synchronous algorithms, that all messages from all nodes has reached all other nodes.

In asynchronous distributed algorithms, nodes may take arbitrary time before producing and sending messages, and the order of events is therefore uncertain. As a result, asynchronous algorithms assume less about time than synchronous algorithms. The challenges that arise in asynchronous algorithms are mainly message pileup and termination conditions. As time can pass arbitrarily before nodes send their next message, incoming messages may pile up. As an example, if a node receives multiple messages before being capable of or deciding to process them and send them, messages may be dropped. This can be handled with i.e. various types of queues, in which FIFO is common. Asynchronous algorithms may not be able to utilise the *diameter* as a guarantee, like synchronous algorithms, as the *diameter* cannot be linked to time or rounds. Instead, asynchronous algorithms have to utilise other techniques for termination conditions. An example, is the use of spanning trees and converge cast. Here, converge casting is

understood as the communication between nodes starting in the leaves of the spanning tree and propagating towards the root. If all nodes wait until they have received messages from all of its children before sending messages to their parent, then the root of the spanning tree can terminate an algorithm after it receives messages from all of its children. Here, termination may have to propagate from the root to all other nodes.

In our UPPAAL models, described in Chapter 6, we create synchronous and asynchronous versions of nodes, to explore possible changes needed to handle asynchronous behaviour, compared to the synchronous version. In our Simulator, described in Section 5.1, we utilise queues to collect different calls from different algorithms, to handle pileup of messages.

4.1.2 Stopping & Byzantine Failures

The book by Nancy A. Lynch [10], also describes two major types of failures that should be considered when dealing with distributed systems. The noted failures are *Stopping failures & Byzantine failures*. *Stopping failures* are any kind of failure that leads to a complete stop of a node. *Stopping failures* can usually trivially be detected, and another node will be able to take over, allowing further operation of the system.

Some failures though, do not lead to a complete stop. This could be a node sending arbitrary messages, taking arbitrary state transitions, or just starting in an arbitrary state. This can lead to unexpected behaviour, which can be hard to detect for the rest of the system. These kinds of failures are called *Byzantine failures*. These errors can be something as simple as a node running out of memory, and as a result, issuing a null response when an actual response was expected. Before the affected node is contacted again, the memory issue could be resolved, and the node could produce the correct response to an identical request. This would lead to the same request producing different responses (null vs. actual) without the node crashing, and as such, be a *byzantine fault*.

In this project, we decided to focus on *Stopping failures*. This was decided in order to prioritise creating an autonomous satellite constellation framework, rather than a framework for handling *byzantine failures*. Detecting and handling *byzantine failures* requires more time and effort, than detecting and handling *stopping failures*. Focusing on *stopping failures* allowed us to focus on development of the features of the distributed algorithms.

In order to be byzantine fault tolerant, a distributed system must be able to decide on a shared consensus regardless of byzantine faults occurring or not.

One of the criteria, noted in [10], is to have $3f + 1$ nodes, with f being the count of byzantine failing nodes. It should be noted though, that the book focuses on byzantine faults in complete graphs. In this project, the graph is more likely to a ring, than a complete graph. Handling byzantine faults in a ring poses multiple problems. Two of these problems will be highlighted below:

Firstly, one problem is in the case of dead node detection. If a node dies, the constellation will build consensus based on statements from the neighbours to the dead node. The first node which detects the dead node will ask the other node for confirmation. In this case, a traitor node will be able to either faultily initiate a failure detection operation, or respond

with a faulty confirmation i.e. confirm a dead node, even if it is not dead or vice versa.

Secondly, if the constellation only allows a node to communicate to its immediate neighbours, messages will have to be passed through nodes from one end of the constellation to the other. One can imagine that a traitor node will be able to modify messages in a way that is not directly detectable by other nodes, or completely discard messages that are not expected by other nodes. This would prevent a message from being propagated to the rest of the constellation, essentially blocking communication between each side of the traitor node. This would prevent the constellation from ever performing a recovery from a dead node, handicapping the constellation. This can be seen in Figure 4.1

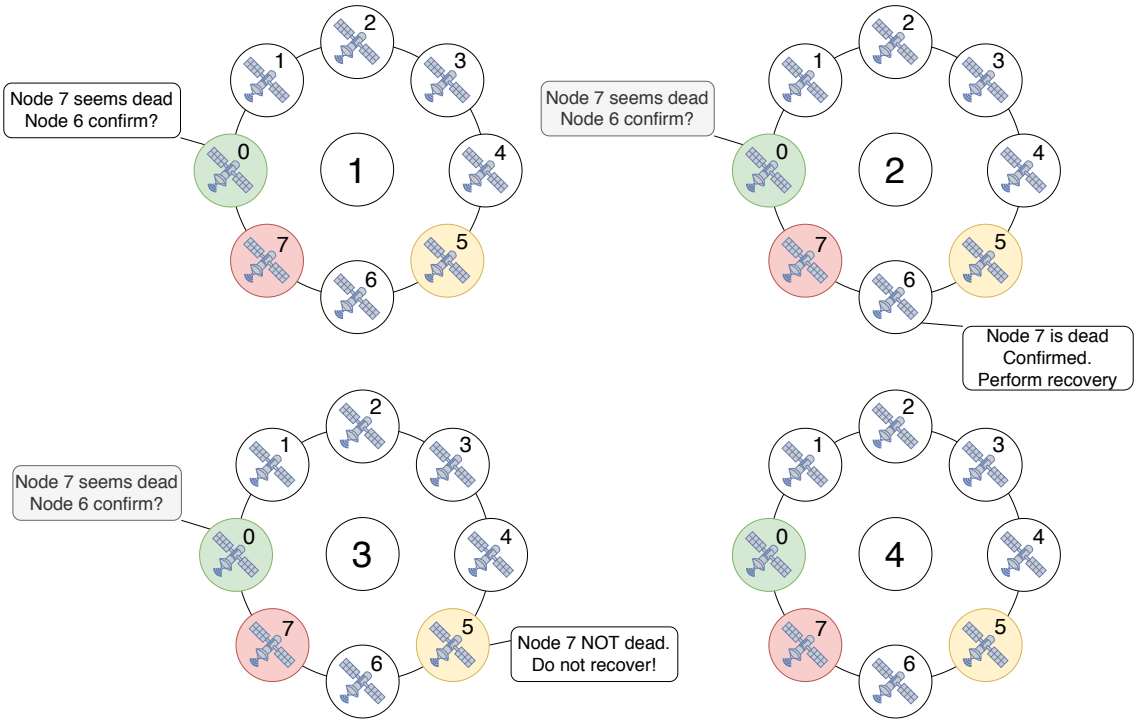


Figure 4.1: Example of byzantine failure. Node 0 initiates a failure detection. Node 7 is the dead node, and node 5 is a traitor node.

In Figure 4.1 we see four snapshots of the same constellation, during a failure detection operation.

In the first snapshot (marked 1), we see the initiating node 0 detecting that node 7 is dead. Node 0 creates a request for node 6, asking for confirmation of the death of node 7, and sends it clockwise towards node 6.

In the second snapshot (marked 2), we see node 6 confirming the death of node 7, which initiates a recovery operation. This message is then propagated counter-clockwise to the rest of the constellation.

In the third snapshot (marked 3), we see the traitor node (marked 5) modifying the message sent by node 6, denying the death of node 7. Node 5 then propagates the modified message to the rest of the constellation, which stops the recovery operation from proceeding.

In the fourth snapshot (marked 4), we see the constellation after the failure detection operation has completed. Since the traitor node (5) stopped the recovery and confirmation of node death, the constellation will not recover from the failure. This breaks synchronisation of the network map of the nodes in the constellation, as well as prevents ring communication. Furthermore, messages sent to or through node 7 will be immediately dropped, as node 7 is dead and cannot receive the messages.

4.2 Architecture

In our framework that facilities our distributed algorithms, satellites are called *Node*. Nodes contain any information about the satellite, such as ID, target position, and state. Furthermore, nodes contain a *Router*, and a *CommsModule*. The router maintains a graph over the network, which is used for calculating which nodes to route through. It also contains a *Network map* used for deducing which nodes can intercommunicate. The network map is also used when planning new constellation plans as it contains information about the location of other nodes. The CommsModule contains a reference to the simulator, allowing it to provide information such as reachable neighbours. The CommsModule also handles sending and receiving messages, as well as storing of any received messages, which can be requested by the node. Nodes also run a *MainThread*. The MainThread is a looping method which runs continually on the node. It will fetch requests at regular intervals from the CommsModule. It will then consider the type of request, and decide to either handle it, or relay it to another node. The relationship between nodes, their router, and their comms module can be seen in Figure 4.2.

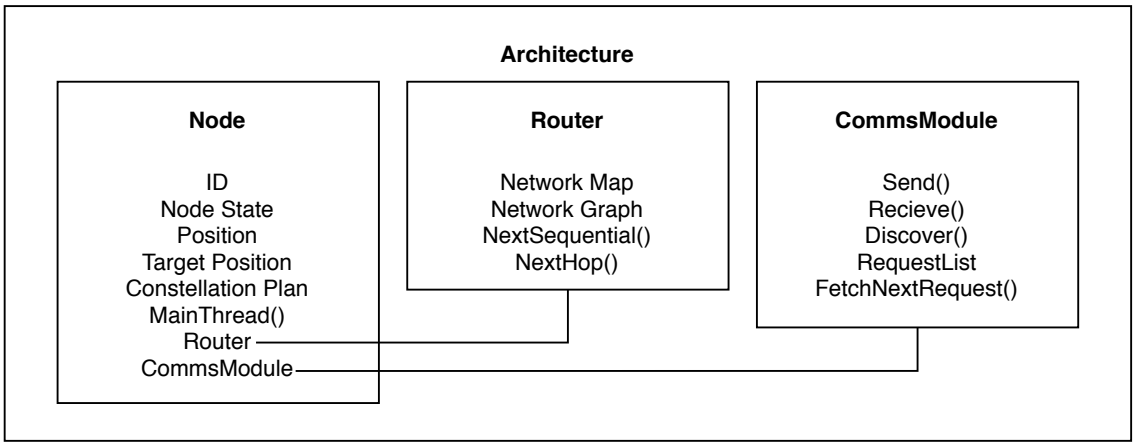


Figure 4.2: Architecture of the satellite nodes

4.3 Plan Generation

Failing satellites can cause holes in the constellation, limiting the communication abilities of the network. In order to repair these holes, the constellation must be able to reorganise, closing gaps and ensuring a continuous networking ring. To facilitate the coordination of reorganising the constellation we create a plan of the new constellation. This plan consists of locations where the satellites should position themselves. It also holds information about which satellite has already signed itself up for a location and how much fuel it must use to

move there. This plan is then propagated around the constellation, each satellite signing itself up for a location each time until all satellites have chosen a location.

Reducing fuel and communication costs in the process of coordinating a plan for a new constellation is of interest. Coordinating which satellite should take which location in a new constellation plan is the responsibility of the plan generation operation. A range of parameters could be taken into consideration while solving this task. Fuel, battery charge, time, and communication to name some. We chose to consider fuel and communication, more specifically how many messages are exchanged amongst satellites during the coordination. Fuel is considered as it is a finite resource limiting the life span of the satellite. Communication is considered as reducing the amount of hops in a path also lowers the transmission time, as well as the risk of package loss to the high loss rate in space compared to on Earth. We simplify how fuel is represented by using distance as the crow flies between the satellite's current location and new locations in the constellation plan. Simplifying fuel requirements disregards the fact that orbital manoeuvres can generally be performed with a high fuel cost and low time cost, or a low fuel cost and high time cost. A satellite would be able to reduce fuel consumption at the cost of time, if it was to be limited on fuel. We choose the goal of the planning logic to be to minimise the total fuel used by the satellites to achieve the new constellation as well as to minimise the amount of messages exchanged. We have experimented with different approaches to solve this task, some requiring more coordination than others.

Figure 4.3 contains a flowchart that shows the general flow of the plan generation approaches.

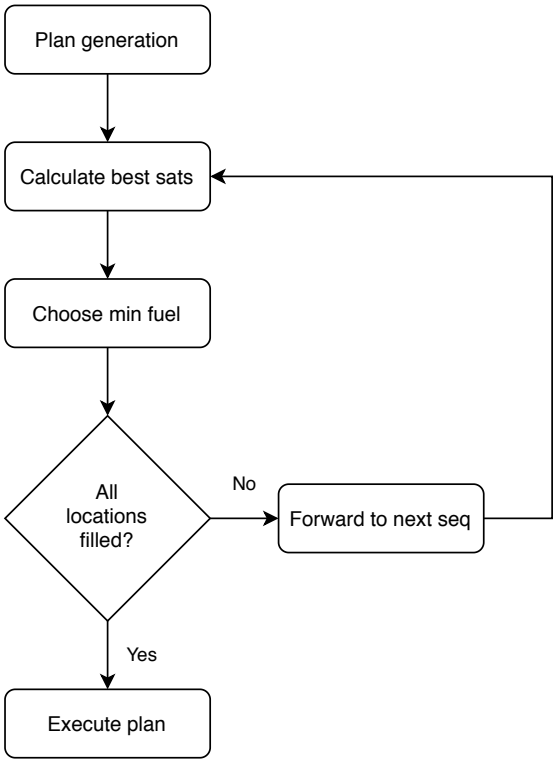


Figure 4.3: Plan generation

The following list elaborates on the steps in Figure 4.3.

- *Plan generation*: The plan generation logic is initiated either by other modules on the same satellite or by receiving a plan generation message from other satellites or from Earth.
- *Calculate best sats*: The satellites that use the least fuel to move to each location is calculated for all locations.
- *Choose min fuel*: In case a satellite is the best one for multiple locations, the location that requires least fuel to move to is chosen.
- *All locations filled?*: Are all locations in the plan filled by a satellite?
- *Forward to next seq*: The plan generation message is forwarded to the next sequential neighbour for further satellites to fill out locations.
- *Execute plan*: Execution of the plan is initiated and an execute plan message is forwarded to the next sequential neighbour.

The first approach we tried was that each satellite greedily took the location that required the least fuel to move to. When a satellite has taken a location in the plan, the plan is forwarded to the next satellite and is propagated around until all satellites have taken a location. The last satellite to take a location then propagates an execute message around so that all satellites execute the coordinated plan. This approach did not yield good results in regards to minimising the overall fuel that the constellation had to use.

Therefore, we tried another approach where each satellite would greedily take the location that required least fuel like before. However, when all satellites had taken a position, a *trading* round would happen where each satellite would see if trading places with another satellite would be cheaper for the constellation regarding overall fuel consumption. This trading round would continue as long as some satellite had swapped locations within a cycle in the constellation meaning that each time a satellite had swapped, the cycle counter was reset having to take another cycle before being able to terminate. This eventually yielded satisfying results, however, it could take many cycles in the constellation before reaching this solution. Furthermore, this approach could lead to an infinite trading loop, where satellites would keep trading and never reach a final state.

In an attempt to minimise the number of cycles around the constellation, yet still get good results regarding fuel, we tried another approach. This time each satellite would sort empty locations from lowest fuel usage to highest and check whether another satellite could take the location cheaper, moving on to the next location if that was the case. This way it could be done in one cycle and satellites would only take locations where they would perform the best.

We tried a similar approach where each satellite would calculate which satellites could fill which locations using the least fuel. The satellite would then choose amongst the ones where it would use the least fuel, while not taking any other node's lowest cost location. This could also be done in one cycle. This approach is the plan generation algorithm that is currently implemented in the simulator.

We also attempted to generate optimal solutions by enumerating all permutations of the satellites on the locations using Heap's algorithm [11]. The number of permutations is exponential in regard to the number of satellites in the constellation. For a small

constellation this might be a feasible solution as it would not take that much time to generate all permutations. However, for a constellation of eleven satellites it took way too long and for larger constellations it would never be a feasible solution.

4.4 Routing

The framework utilises two different modes of routing. The different modes come into play depending on the situation. If a message is only intended for a single node, shortest path will be utilised. If the message has to be seen by every node on the way, Radial Sweep, see Subsection 4.5.2, will be used. In routing along the entire constellation, messages will be passed along to direct neighbours. These direct neighbours will be known in the report as *sequential neighbours*, where a *next sequential neighbour* is the sequential neighbour in the communication direction.

4.5 Discovery

Since the satellites in a constellation do not have perfect global knowledge, some form of synchronisation is required in order to obtain and propagate information. If a satellite has been added, moved or removed, this knowledge will need to be known by other nodes. This is the task of the Discovery operation. The operation utilises passes in the constellation, allowing each node to add any new information to the request, and then continue the propagation. The result of a discovery operation will be either an updated network map for each node in the constellation, or a new type of operation getting executed.

The operation will alter behaviour based on what it encounters. If it encounters a new node, not currently in the network map, it will produce a network map addition. This contains information of the new node, and will be utilised by the proceeding nodes in implementing the new node into their own network map. If the operation encounters a dead node, a heartbeat operation, see Section 4.6, will be executed, which will take over and determine the next course of action. In order to minimise the amount of messages relayed in the constellation, discovery will only propagate if any new knowledge is found. This means that if a node does not gather any new knowledge about the network topology from the received request, the discovery request will be dropped, and the operation stopped. This is also the case if the node has not found any new local information about its vicinity, regarding the network topology. One thing that can force the constellation to proceed with a discovery operation, regardless of the addition of knowledge, is the *Require Full Sync* flag. If this is raised, the constellation will be forced to propagate the message along a given direction until the end has been reached. The operation will then reverse the communication direction, and propagate the request until the other end has been reached. Lastly the ending node will decide whether to take action or not. The *Require Full Sync* flag will be used after a recovery procedure has been executed, and new nodes have been discovered. This is one of the primary parts of the self healing properties of the constellation framework. The flow of the discovery operation can be seen in Figure 4.4.

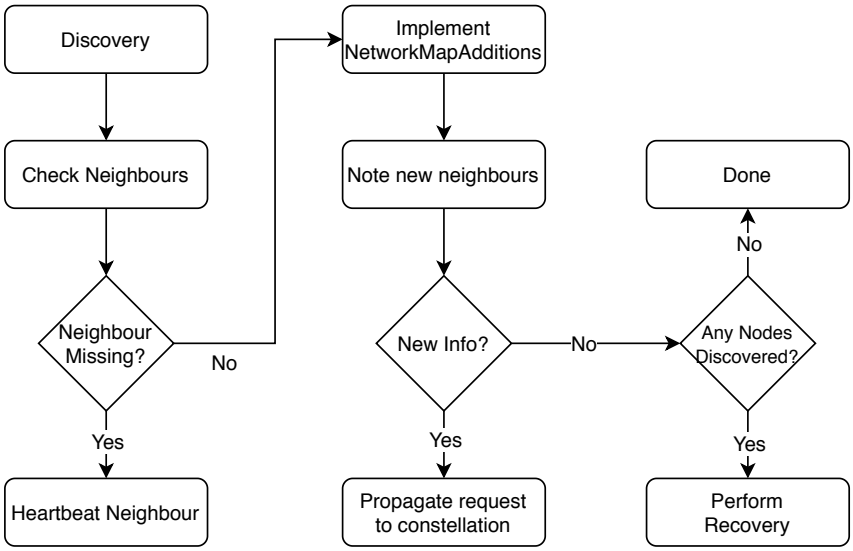


Figure 4.4: Discovery Flow

The following list elaborates on the steps in Figure 4.4.

- *Discovery*: This is the initial step. This starts the operation
- *Check Neighbours*: Take note of the current neighbours
- *Neighbour Missing?*: Check if any of the previously known neighbours has gone missing
- *Heartbeat Neighbour*: If one or more neighbours have gone missing, discovery will initiate the heartbeat operation, which will in turn decide if a failure detection operation should be executed
- *Implement Network Map Additions*: If no neighbours have gone missing, the node will implement any network map additions from the request into its own network map.
- *Note New Neighbours*: Add any new neighbours detected in *Check Neighbours* step to the request.
- *New Info?*: Check if the request contained any new information the node was unaware of, or the node had new information the request was not aware of, such as a new neighbour. Also check if the discovery request requires full sync. If any of these are true, *New Info* will be true.
- *Propagate Request to Constellation*: Pass the request onto the next node (see Figure 4.5).
- *Any Nodes Discovered?*: Check if the request has any Network Map Additions (If any node at any time in the discovery operation has noted new nodes).
- *Done*: The operation has been completed. Return to main thread and take on any new requests.
- *Perform Recovery*: Perform a recovery operation, in order to implement the new node(s) into the constellation. (See Figure 4.10).

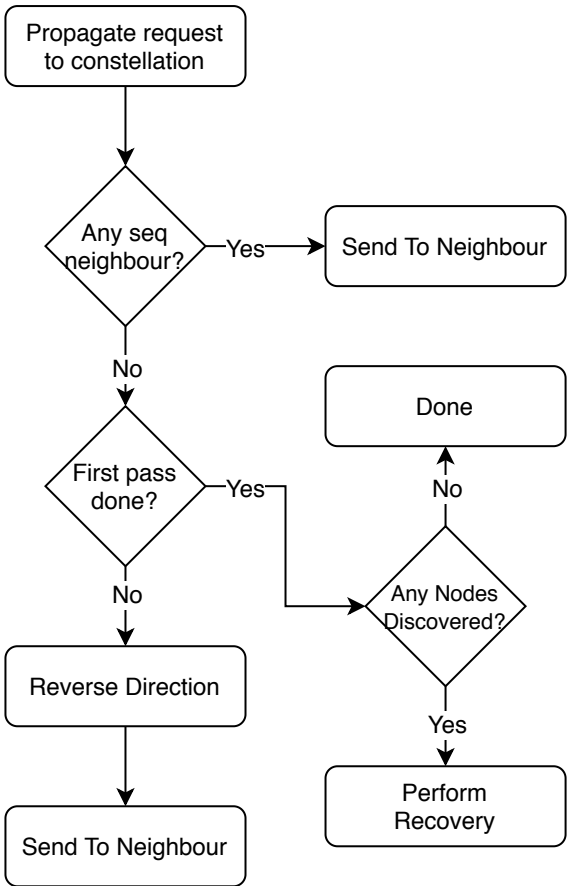


Figure 4.5: Discovery Propagation used by the discovery operation seen in Figure 4.4.

The following list elaborates on the steps in Figure 4.5.

- *Propagate Request to Constellation*: The initial step. This starts the operation.
- *Any Seq Neighbour?*: Check if any neighbour can be reached in the current propagation direction.
- *Send to Neighbour*: Pass the request on to the next sequential neighbour.
- *First Pass Done?*: Check if the request has reached the end of the constellation in the initial direction.
- *Any Nodes Discovered?*: Check if the discovery request has found any new nodes
- *Done*: The Discovery request has found no new nodes to add to the system. The operation has finished.
- *Perform Recovery*: Perform a recovery operation, in order to implement the new node(s) into the constellation. (See Figure 4.10).
- *Reverse Direction*: Reverse the direction of the request, making it pass along the constellation in the opposite direction in order to find the other end of the constellation.
- *Send to Neighbour*: Pass the request on to the next sequential node in the request direction.

The following figures will display functionality of the discovery operation in a couple of different example scenarios.

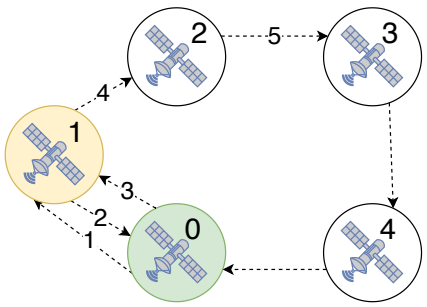


Figure 4.6: Discovery Addition. Node 0 is the initiating node, Node 1 is a new node.

The following describes the flow in Figure 4.6:

- *1*: The initiating node begins a *Discovery* operation, detecting the new node. The initiating node will then send a Network Addition Request to the new node, informing it of the current network map, as well as requesting information such as position from the new node
- *2*: The new node will inform the initiating node about any requested information.
- *3*: The Initiating node performs a *Radial Sweep* (see subsection 4.5.2) and finds the new node to be the next sequential. The initiating node will then pass the Discovery request on to the new node.
- *4+*: The discovery request will be propagated clockwise to the rest of the constellation, until it eventually reaches the initiating node, where a *Plan Generation* operation will be initiated (see Figure 4.3) in order to add the new node to the constellation.

When a constellation plan has been executed, nodes can start a discovery operation if they register new neighbours that are not part of the constellation plan. This functionality is used when a constellation has been divided into two or more sub constellations, and functions to merge any detected sub constellations back into one constellation.

4.5.1 Topology

Every node in the constellation will keep note of every satellite in range. This will define which satellites can be communicated with. If possible, each satellite will define a next sequential satellite in each direction, clockwise and counter-clockwise. Next sequential nodes are immediate neighbours, and can be communicated with, without skipping any other nodes. The next sequential neighbours will be used when a message has to be passed through all satellites on the way. Shortest path is allowed to utilise other links, in order to minimise the amount of hops in the path.

Figure 4.7 displays an overview of a rather chaotic network. In this network, we can see the ring topology, marked with a solid line, with the rest of the links marked with a dotted line. Here, it is clearly visible that the constellation has established a ring.

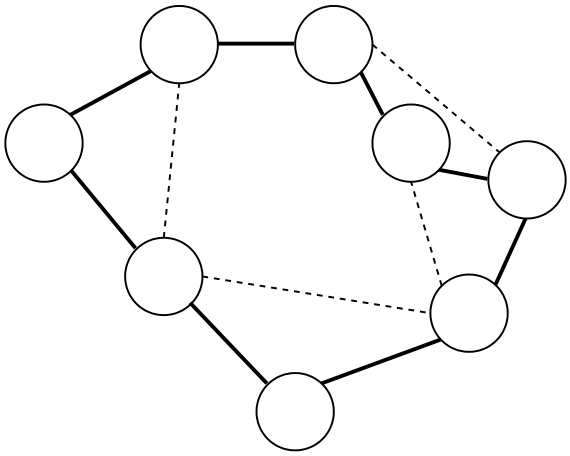


Figure 4.7: Network Topology Example

4.5.2 Radial Sweep

In satellite constellations, some messages have to be propagated to the entire constellation. In order to ensure that all nodes would receive a message, and try to lower the amount of hops needed for a complete pass in the constellation, we decided to implement a radial sweep algorithm. This algorithm is used when deciding on the next node in the propagation.

Using radial sweep, the constellation will, no matter the actual geographical topology of the constellation, communicate in a ring. This is done by performing a radial sweep around the centre of the constellation. Radial sweep consists of two phases. Initially, a direction vector will be created for each node, from the centre of the constellation. The executing node will compare every vector to its own vector, and decide the next sequential node based on communication direction. For clockwise communication, the lowest positive angle will be chosen. For counter-clockwise the highest negative will be chosen. This ensures that, if possible, no nodes will be missed in the constellation, and complete cycles actually contain all nodes. If the next sequential node cannot be reached directly by the sending node, it will be routed by means of Dijkstras shortest path algorithm, to the next sequential node.

As seen in Figure 4.8, the initiating node marked 0 will decide the angles of any other node in the network map, with respect to the normal vector of the constellation plane. This ensures that no node, that is known to the calling node, will be skipped.

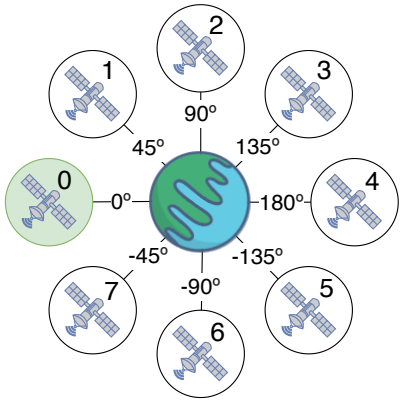


Figure 4.8: Radial Sweep algorithm displaying initiating node 0 as green, next sequential clockwise node 1, and next sequential counter clockwise node 2.

4.6 Heartbeat

The heartbeat operation is responsible for contacting neighbours regularly and triggering fault handling in case no communication can be established. The goal of the heartbeat operation is to trigger fault handling in the case that a neighbour cannot be successfully contacted.

Communication with a satellite's neighbours is regularly triggered with some delay. Contacting the neighbours is attempted at most three times with increasing delay between these attempts, before triggering fault handling, see section 4.7, if the link is failing. The number of communication attempts can be set based on one's requirements. The heartbeat operation sends a ping message to the satellite's neighbours and expects a ping response within a user defined time frame.

4.7 Fault Handling

The fault handling operation is responsible for detecting failed satellites and maintaining a constellation in which all active satellites are connected. The goal of the fault handling operation is to reconfigure the constellation so that a fully connected ring is established. The fault handling logic must be able to handle different patterns of failed satellites, namely, sequential failed nodes and non-sequential failed nodes. The fault handling logic is successful if it can handle both of these patterns.

A flowchart illustrating the overall idea of the failure detection logic can be seen in Figure 4.9.

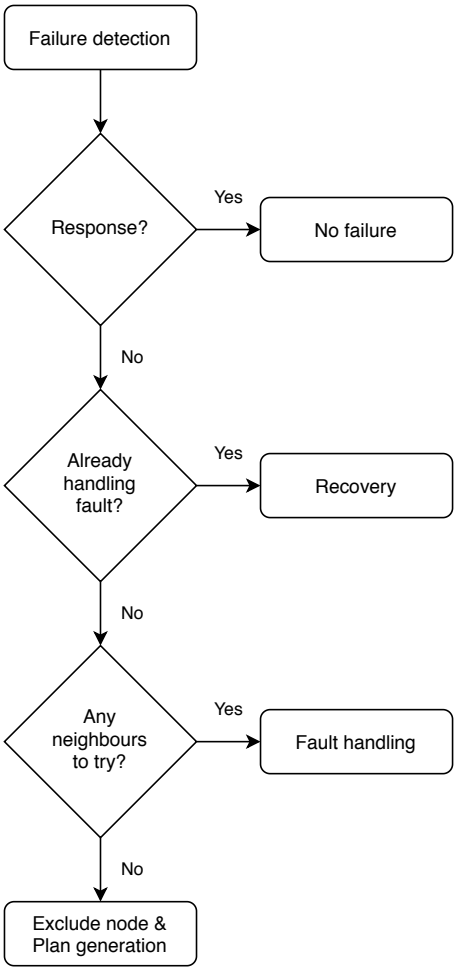
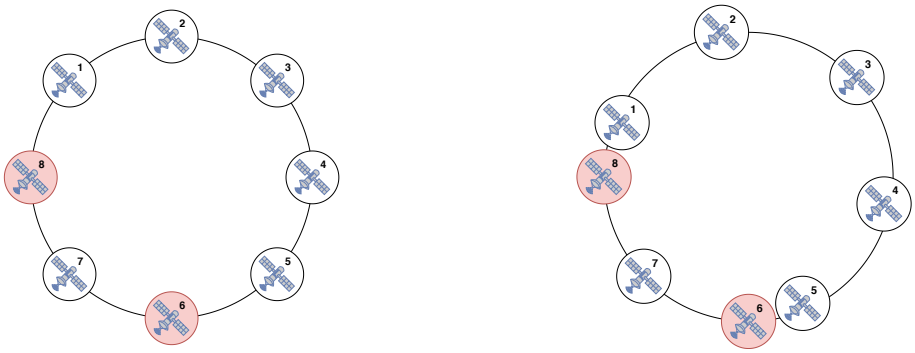


Figure 4.9: Failure detection

The following list elaborates on the steps in Figure 4.12.

- *Failure detection*: The failure detection logic is run for all messages that require an acknowledgement of reception.
- *Response?*: Was an acknowledgement received within several attempts of sending the message?
- *No failure*: No failure is detected and execution is continued.
- *Already handling fault?*: Is the message that was attempted to be sent about another failure?
- *Recovery*: A recovery plan with locations for the nodes that is part of the current node's network is generated. These locations are placed so that the nodes are fanned out as much as possible while still being within communication distance (see Figure 4.10).
- *Any neighbours to try?*: Does the seemingly failed node have any neighbours that could try to contact it?
- *Fault handling*: The fault handling logic is run when there are neighbours to the failed node that can try to contact it (see Figure 4.12).
- *Exclude node & Plan generation*: A constellation plan without the failed node is generated and propagated (see Figure 4.11).

Recovery is performed when no acknowledgement is received while attempting to send a message and this message is already about another failure (see Figure 4.10).

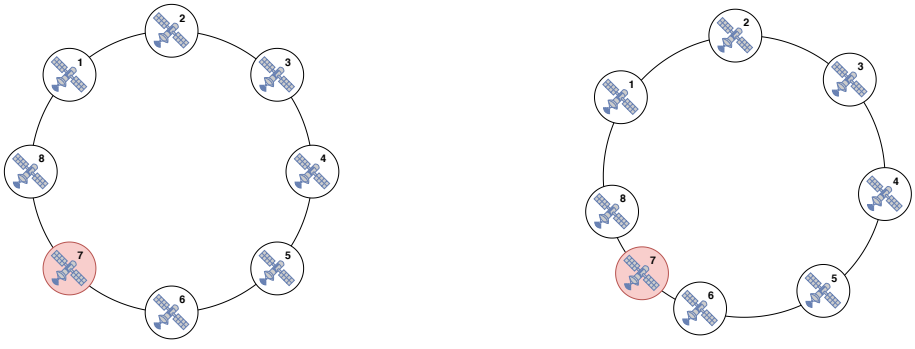


(a) Before recovery (b) After recovery

Figure 4.10: Recovery. The numbers are IDs of the satellites. The red satellites (6 and 8) are failed satellites.

A constellation in a recovery situation is seen in Figure 4.10a. Satellite 1 has tried to communicate with 8. Satellite 8 did not respond with an acknowledgement within several attempts. 8 has a neighbour (7), so 1 forwards the fault handling to satellite 7 routing via 2 through 6. However, when satellite 5 tried to communicate with 6 it did not respond with an acknowledgement. 5 then initiates recovery by generating a recovery plan for the satellites it is connected with (satellite 1 through 5). 1 through 5 coordinate which satellite takes which location in the recovery plan and it is ultimately executed. The constellation ends up looking like the one seen in Figure 4.10b. Satellite 1 through 5 have fanned out in an attempt to reconnect with satellites from the other sub-constellation, in this case there is only one (7).

Exclusion of a node happens when none of its neighbours can communicate with it (see Figure 4.11).



(a) Before exclusion (b) After exclusion

Figure 4.11: Exclusion. The numbers are IDs of the satellites. The red satellite (7) is a failed satellite.

A constellation before exclusion is illustrated in Figure 4.11a. Satellite 8 tries to communicate with 7 but without luck. 7 has a neighbour (6) which satellite 8 forwards

the fault handling to via 1 through 5. Satellite 6 has no luck contacting 7 either thus generating a new constellation plan excluding satellite 7. The plan is coordinated amongst 1 through 6 as well as 8 and executed. This results in the constellation in Figure 4.11b where a fully connected constellation is achieved now without satellite 7.

Figure 4.12 contains a flowchart that shows the general logic of fault handling.

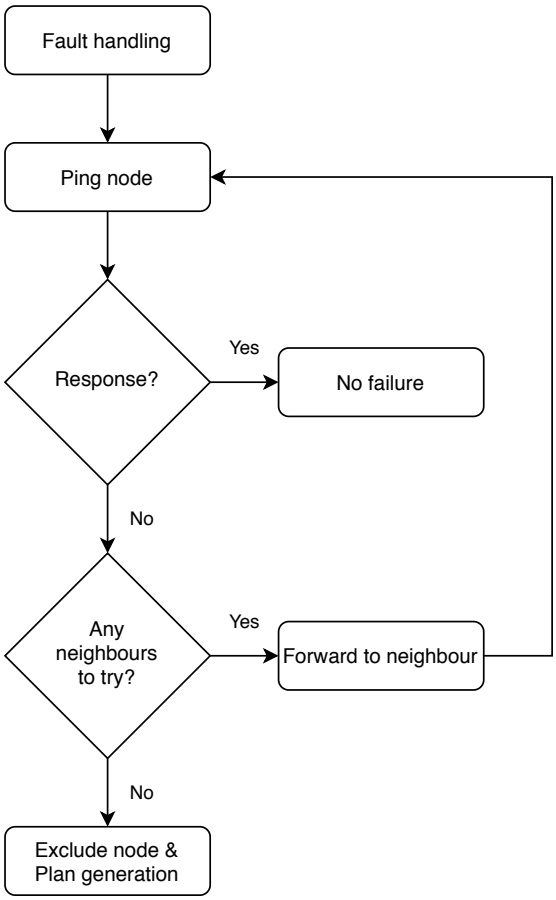


Figure 4.12: Fault handling

The following list elaborates on the steps in Figure 4.12.

- *Failure handling*: The fault handling logic is run by the failure detection when there are neighbours to the failed node that can try to contact it.
- *Ping node*: The failed node is pinged several times in case no response is received right away.
- *Response?*: Was an acknowledgement received within several attempts of sending the message?
- *No failure*: No failure is detected and execution is continued normally.
- *Already handling fault?*: Is the message that was attempted to be sent about another failure?
- *Any neighbours to try?*: Does the seemingly failed node have any neighbours that could try to contact it?
- *Forward to neighbour*: The fault handling is forwarded to another neighbour node and the process is started over at that node.

- *Exclude node & Plan generation*: Finally, if there are no more neighbours to try then a constellation plan without the failed node is generated (see Figure 4.11).

An alternative to deeming nodes failed after several attempts at communicating with it within a relatively short time frame could be to define some threshold that could be configured after one's specifications. The node would be deemed failed if no successful contact was made with it within this time frame, however, until the threshold was met neighbours would try with regular intervals to contact it, possibly reaching it at some point.

Visual Simulation 5

This chapter introduces and explains the simulator application and briefly explains how to use it. It also introduces the Unity engine that the simulator is built in, in which we reflect on the challenges we met building the simulator.

The simulator is the second contribution in the project. It provides a fully functioning host for the use of the constellation framework. In the simulator, the user has the ability to interact with the virtual constellation in different ways, each described below:

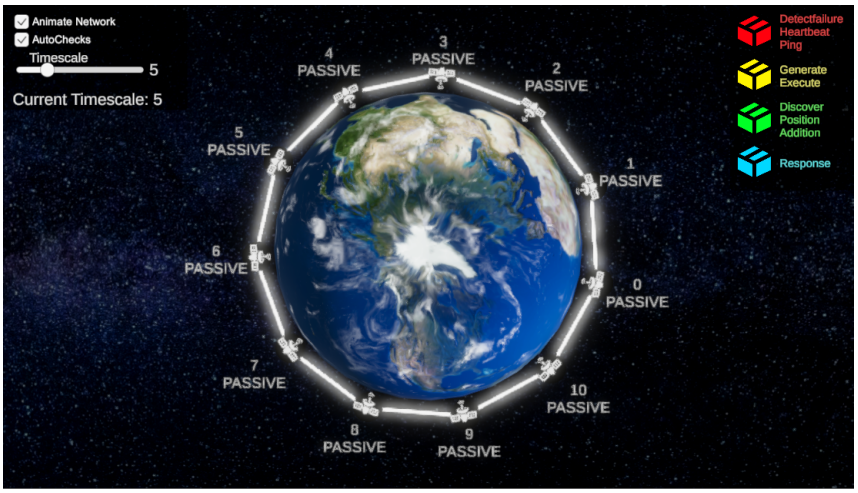


Figure 5.1: Initial state of the simulator, displaying a constellation consisting of 11 nodes.

- *Add Satellite* is activated by right clicking empty space in the simulator and will add a new satellite into the simulation space.
- *Open Context Menu* is activated by right clicking any satellite and opens a context menu for the given satellite, allowing different actions. The context menu can be seen in Figure 5.2
- *Context Menu* → *Toggle* will toggle the state of the node. If the node is active, it will be instantly disabled and vice versa. This makes the node non-responsive to other nodes, and functions as a stopping failure.
- *Context Menu* → *Discovery* initiates a *Discovery* operation on the chosen node.
- *Context Menu* → *Heartbeat* initiates a *Heartbeat* operation on the chosen node.
- *Context Menu* → *Generate* initiates a plan generation operation on the chosen node.
- *Context Menu* → *Remove* Removes a node, and any reference of it in other nodes. Making it as if the node never existed in the first place.
- *Top Left Menu* → *Animate Network* toggles animation of the constellation, deciding if it will rotate around the earth or not.

- *Top Left Menu → Auto Checks* toggles whether nodes should run periodical heartbeat and discovery requests.
- *Top Left Menu → Timescale slider* requests a change in timescale. The change may not be instantaneous, as the framework will synchronise timescale of all nodes.
- *Design constellation* is done by left clicking empty space after a plan generation operation has successfully concluded. This will place *locations* that the constellation must fulfill in order to complete the constellation. When the correct amount of *locations* has been placed, the new constellation Plan will be sent to a random node, which will be tasked with handling the request propagation.

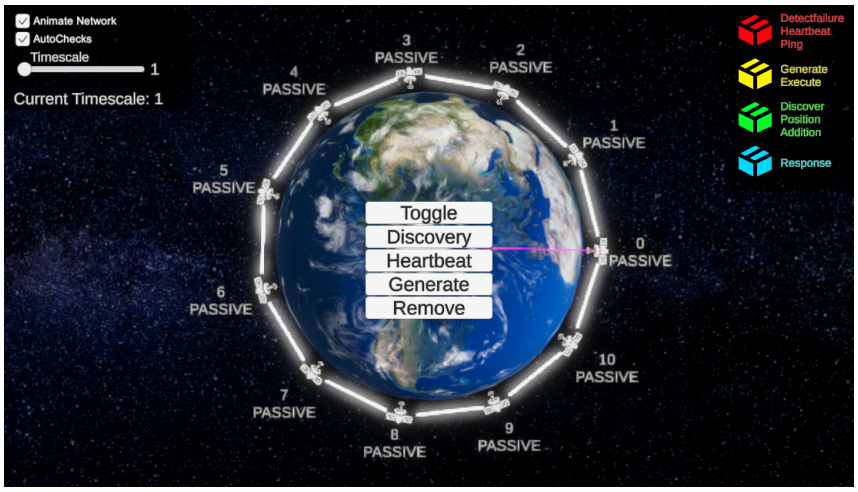


Figure 5.2: Contextmenu of simulator shown for node 0, after right clicking the node

Furthermore the simulator will display a multitude of information providing it to the user. The displayed information is explained below:

- *Satellite position* The current position of the satellites will be displayed, with relation to the earth and the rest of the constellation. This also includes the simulation of satellite movement when transitioning from one location to another.
- *State & Operation phase* The simulator will display the current state of the satellite, whether it be *DEAD* or any operation such as *DISCOVERY*, allowing the user to see exactly what each satellite is doing in the constellation.
- *Communication* Reachable satellites will be displayed by a thin white line, while next sequential satellites in both directions are displayed by thick white lines. Furthermore, messages sent are shown as coloured boxes, these boxes are coloured based on the type of request. Requests related to failure detection are coloured red, constellation planning are coloured yellow, discovery are coloured green, and responses are coloured cyan.
- *Locations* During plan generation and constellation design, the desired locations for the constellation will be displayed as green circles.
- *Satellite Desired Location* During plan generation, the simulator will display which locations are desired by which satellites. This is displayed as a yellow line between the satellite and the location.

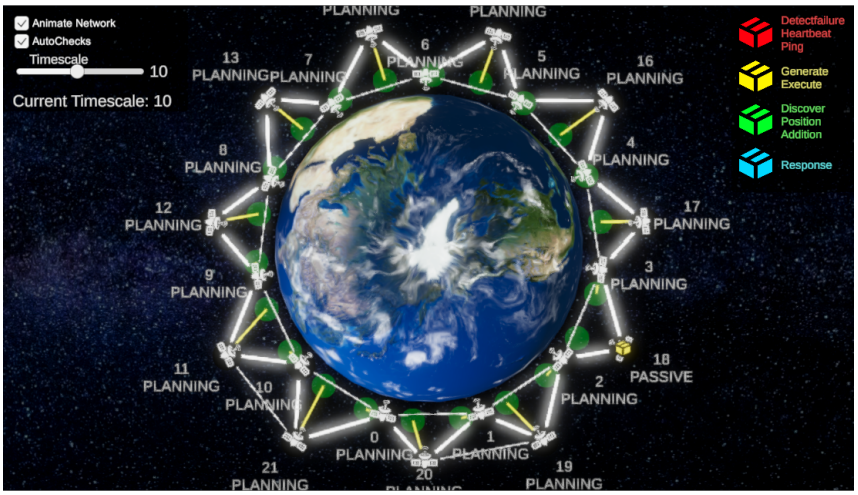


Figure 5.3: Plan generation phase of the constellation displayed by the simulator, after a discovery has detected the outer nodes.

Since the framework is run in a virtual environment, and not on actual physical hardware, the simulator also emulates some of the required information, which can be seen in Figure 5.4. For one, it will supply the framework with node positions. This means that the simulator will be moving the nodes, and inform the framework of the movement. The simulator also has a communication module simulator, which emulates the role of a physical communication module, and is tasked with sending and receiving messages, as well as detect which nodes are reachable by its owning node.

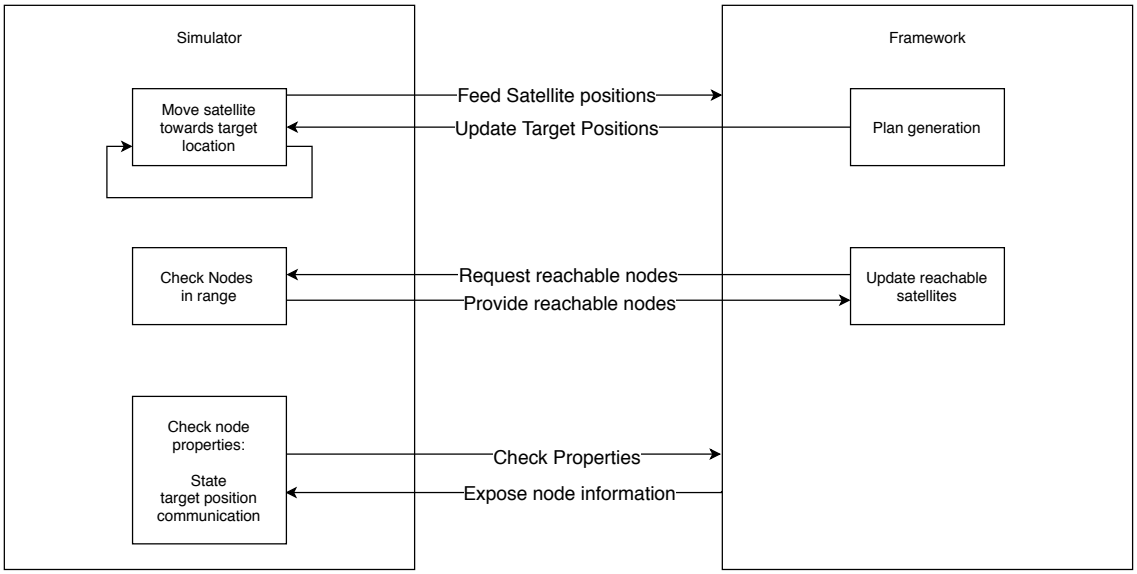


Figure 5.4: Relation between simulator and framework

5.1 Unity Engine

The Unity Engine is a cross-platform game engine, which allows fast and easy building for multiple platforms, as well as provide a framework and visual editor for producing games and simulations. In this project, the Unity engine was chosen based on the amount of experience the group members had with the engine, as well as the excellent tool set Unity provides. Other 3D simulation software may be better suited for the task, however, we went with Unity as we all had experience with it allowing us to get started right away.

In this project we decided to run each satellite in its own thread. This was done for two reasons. The first being that we wanted to detach the framework from Unity as much as possible. We did this so that the satellite constellation logic can run on its own and is not tied together with nor dependent on the simulation. The simulator, however, does feed the satellites with position data but this would be swapped out for an inertial navigation system or GPS in a realistic setting. The second reason was to prevent blocking the main thread, which would result in lowered frame rates, or the simulator completely freezing as Unity is built to run in a single thread. In order to utilise the Unity API, any API calls must be done in the main thread of Unity. Furthermore, any exceptions thrown in any other thread than the main thread of Unity will not be caught by the debugger. This has resulted in a variety of different issues during development, where long stack traces had to be followed by hand in order to determine the faulty line of code.

Furthermore, since the simulator is only supposed to display single-plane constellations, implementation of operations in the framework was prioritised over performance of the simulator. This led us to over-utilise the *Update* method in the Unity API. This method runs every frame in the simulation, and as such, is easy to use. One of the most expensive components in the simulator is called "ConstellationVisualiser". This is tasked with displaying changes to the constellation, such as node state, sent messages, and communication lines. This functionality could be more optimally implemented as an event system, allowing the code to only run when it is required, and not every frame.

UPPAAL Models & Queries 6

In this chapter, we present UPPAAL models of our distributed algorithms and discuss how we use UPPAAL's verification. We also discuss how we have explored state space explosion and verification time, and how we have used UPPAAL to test variations of our algorithms.
We use UPPAAL to test that our algorithms always behave correctly, terminate, and give guarantees of how much communication is required to solve the problem. We verify this for both synchronous and asynchronous environments. Here, correctness is understood as nodes producing a correct result, either individually or combined. As an example, in *Plan Generation*, see Section 4.3, a correct result is that every node chooses a distinct target position as their new position.

We leave out details or assume they are known from the start if they are not necessary for the distributed aspects of our algorithms. We do this to reduce the complexity of our models and their verification time. As an example, in the *Discovery* algorithm, see Section 4.5, each node searches the space around it and discovers new nodes that are not part of the current network. In our *Discovery* model, we assume the new nodes are known from the start. As a result, we verify the distributed aspect of synchronising the knowledge of the network, and leave out the details of how each node individually discover new nodes. We make sure to point out when our models leave out or assume prior knowledge.

All of our models consist of a synchronous and asynchronous node template and one or several setup templates. The node templates implement the algorithms running in each node and the setup templates initialise the scenarios in which the nodes have to perform their algorithms. In the synchronous nodes, each transmitting action takes exactly one time unit, whereas it can take arbitrary bounded time units in the asynchronous nodes. Algorithms that work in synchronous environments may not produce correct results in asynchronous environments, with the same logic. We use the two versions of our node templates to check if they work in both environments, or how they can be modified to work in asynchronous environments. We also make sure to point out differences in our models and algorithms, in order to make them work in asynchronous environments.

6.1 Plan Generation Model

The Plan Generation model consists of two setup templates that create two types of scenarios, as well as a synchronous and asynchronous node template. We model two types of scenarios, to verify that Plan Generation works correctly in both. The node template models a satellite with the Plan Generation algorithm running. Here, each node is responsible for choosing a target position when it receives a transmission, and transmits to its next neighbour or sends an Execute message to finish the Plan Generation. We verify that all nodes choose a distinct target position, and that they all terminate after Plan Generation ends. We also check the communication complexity.

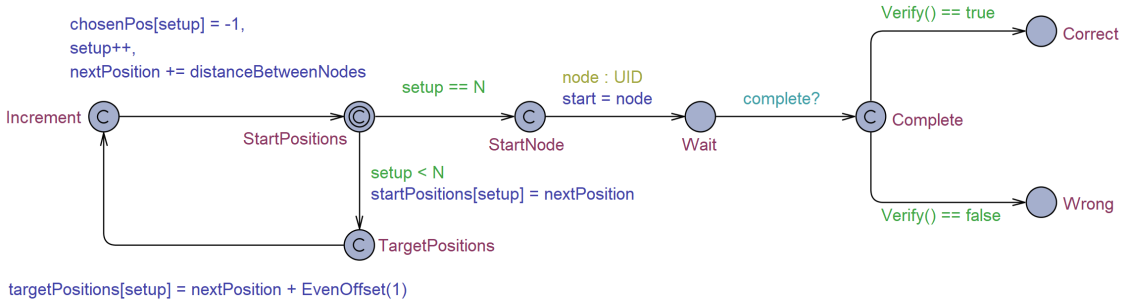


Figure 6.1: Plan Generation Setup - Evenly Spaced Nodes. Nodes and target positions are placed in the loop starting in the *StartPosition* location. Nodes are placed evenly spaced and target positions are placed between two nodes, with a possible offset. The offset is selected from the *TargetPosition* location. Lastly, the node that should start Plan Generation in selected in the *StartNode* location.

The plan generation setup templates can be seen in Figure 6.1 and Figure 6.2. Figure 6.1 creates node start positions that are evenly spaced and target positions that are spread out between each node as seen in Figure 6.3a. The selection in the outgoing edge in *TargetPosition*, places each target position either in the middle of two nodes or closer or further away from the last node placed. Note, that there is exactly three choices and the close and further away positions are always with the same offset from the middle. As a result, when we verify our algorithm with this setup template, we verify all possible combinations of offset position. This is also the reason to limit it to three choices, otherwise the number of possible variations to the scenario would increase dramatically and as a result also dramatically increase verification time. Figure 6.2 creates start positions as if a node has died, creating a hole in the original setup, and creates target positions, that may be offset from being evenly spaced. As a result of the potential extra offset to target positions this setup may create target positions, in which there is no target position between two nodes, as seen in Figure 6.3c.

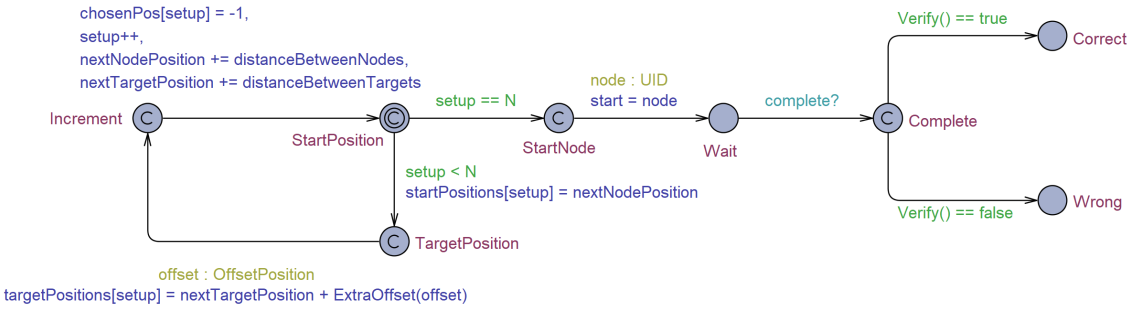


Figure 6.2: Plan Generation Setup - Failed Node. Here, nodes are evenly spaced as if it was a complete ring, but the first node is not placed, to emulate a ring with a missing node. Target positions are evenly spaced, again with a possible offset. Here nodes and target positions are placed separately, as seen in the changes from the *TargetPosition* and *Increment* locations.

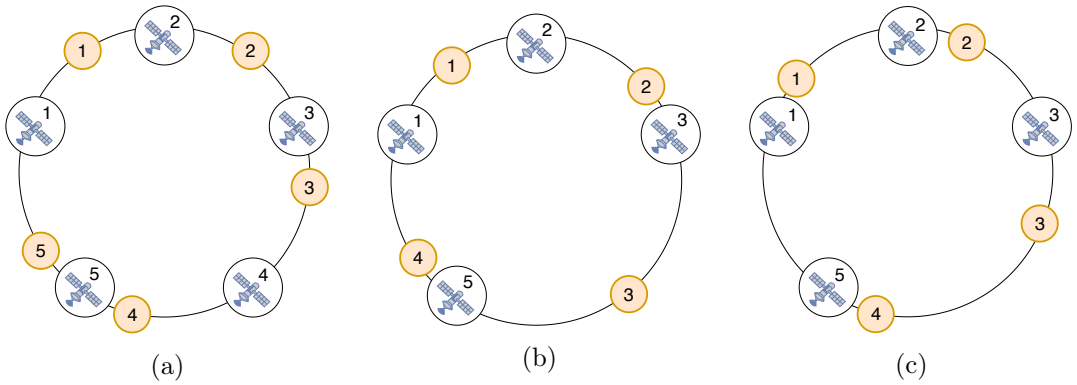


Figure 6.3: Plan Generation Scenarios. Numbered circles represent target locations. *(a)* shows evenly spaced nodes, with target positions either being in the middle of two nodes (1 & 2) or closer to one (3, 4 & 5). *(b)* shows the same nodes where node 4 has died, creating a hole, and the target positions being evenly spaced. *c* also has a hole in the constellation, but with no target position between node 1 and 5.

To model that the nodes and target positions are in a ring we use a line. When we calculate the cost, which is the distance between two positions, we simply wrap the end and start of the line to imitate a circle. We tried using vectors to represent positions, but was unable to create a setup template that was able to place positions in a circle. This is due to UPPAAL being unable to handle *doubles* in certain situations. We tried to work around this problem by only handling and creating *doubles* inside functions and converting them to *ints* before returning, in order to not have *doubles* in the state space. However, this did not solve the problem and we opted to use a wrapping line to represent the ring.

Both setup templates are also responsible for choosing one node as the starting node to verify that the algorithm works in all scenarios with every node being the starting node. Lastly, both setup templates are notified when all nodes have terminated and verifies if the result is correct. Correctness in this algorithm is that all nodes have chosen a target position, and that they have chosen a distinct position. The scenarios created in our Plan Generation model do not reflect how new target positions are created in our simulator. We

modelled these scenarios in order to check that the Plan Generation algorithm is robust and works for scenarios that we may not encounter in the simulator.

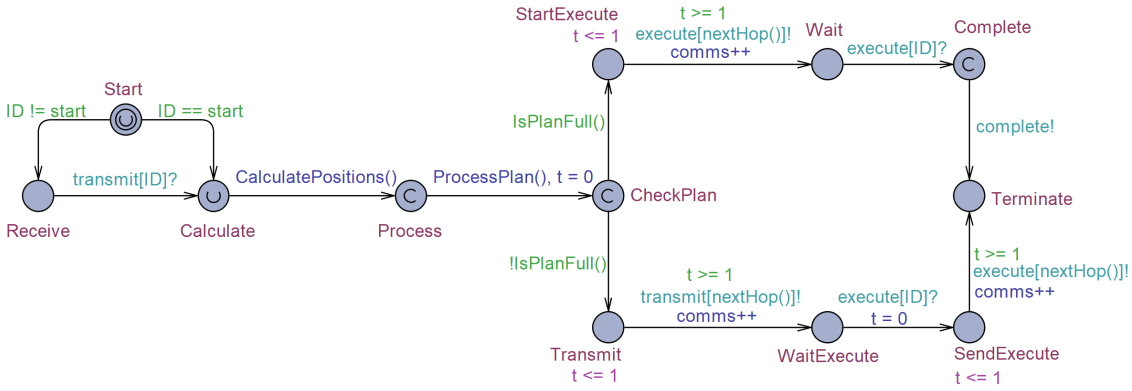


Figure 6.4: Synchronous Plan Generation Template. Nodes receives a Plan Generation transmission in the *Receive* location and choose a target position in the *Calculate* and *Process* locations. The starting node skips the *Receive* location and immediately chooses a target position. Nodes checks if the plan is full in the *CheckPlan* location. If the plan is not full, a node transmits Plan Generation to the next node and waits for an Execute transmission, otherwise it starts an Execute transmission

The synchronous and asynchronous node templates can be seen in Figure 6.4 and Figure 6.5, respectively. The only difference between these two versions, is that the asynchronous model can take between 1-10 time units before each transmission, as seen in the invariant in the *Transmit* location and the outgoing edge's guard. As this algorithm works like passing a token around, the asynchronous environment does not change the behaviour of the algorithm, it simply increases the total time, and no other changes to the template is needed to produce a correct result. The communication around the ring terminates when the node that started the execution transmission receives an execute transmission and other nodes may terminate after transmitting execute to its neighbour.

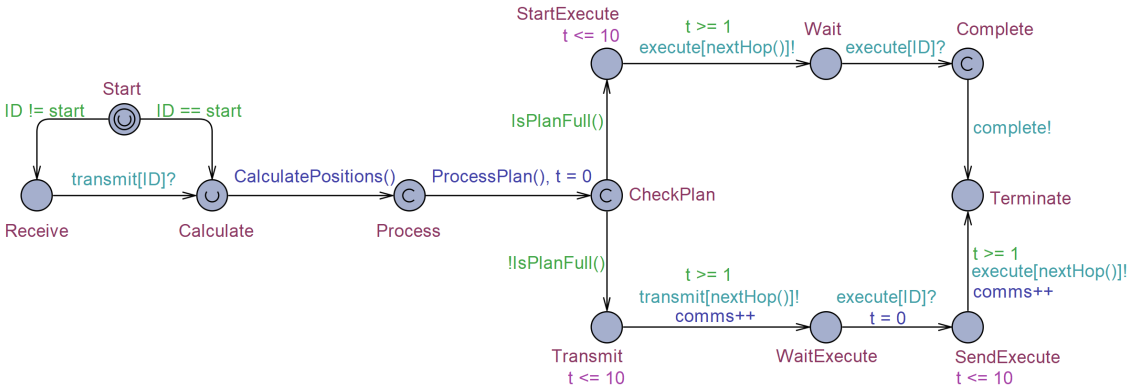


Figure 6.5: Asynchronous Plan Generation Template. Compared to the synchronous version, transmissions may take 1-10 time units.

```
1  /* Termination Queries */
2  A<> forall (i : UID) RingNodeSync(i).Terminate
3  A<> forall (i : UID) RingNodeAsync(i).Terminate
4
5  /* Corretness queries */
6  A<> PlanGeneration.Correct
7  A<> PlanGenerationHole.Correct
8
9  /* Communication complexity queries */
10 A[] comms <= (N * 2) - 1
11 E<> PlanGeneration.Complete && comms <= (N * 2) - 2 // fails
12 E<> PlanGenerationHole.Complete && comms <= (N * 2) - 2 // fails
```

Listing 6.1: PlanGeneration Queries

Listing 6.1 shows the queries that verify the Plan Generation model. Line 2-3 verify that all nodes eventually terminate. Line 6-7 verify that the algorithm is correct in both scenarios. Note that, we change between the synchronous and asynchronous nodes and the evenly-spaced and hole scenario in UPPAAL to verify that both versions are correct in both scenarios. Line 10 verifies that the algorithm at most communicates $(n \cdot 2) - 1$ times for a given n. Here n is the number of nodes. Line 11 and 12 check if there is any trace in which the algorithm communicates $(n \cdot 2) - 2$ or less times. This query is not satisfied and in combination with the query in line 12, we verify that the algorithm communicates exactly $(n \cdot 2) - 1$ times. In the first round of the Plan Generation algorithm, nodes choose a target position. The node that chooses the last target position starts the *execute* transmission. At this point all links except the link between the start node and the node that starts the *execute* transmission has been used, which is $n - 1$ transmissions. All links need to be used before the Execute transmission reaches the node that started it, which is n transmissions, which results in $(n \cdot 2) - 1$.

6.2 Discovery Model

The Discovery model, like the Plan Generation model, also consists of two setup templates, and a synchronous and asynchronous version of the node template. The two setup templates model two scenarios, one where the ring of nodes is complete, and another where there is a broken link. The node template models a satellite that is running the Discovery algorithm, see Section 4.5, in which nodes discover new nodes, and share the new knowledge to the rest of the network. We verify that, after each node is finished with Discovery, all nodes have included the new nodes and new edges.

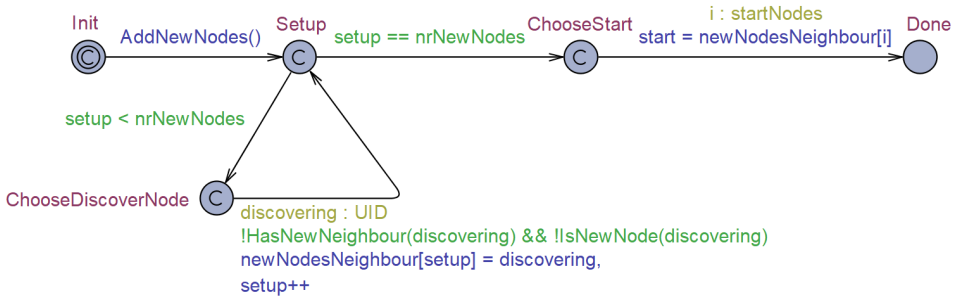


Figure 6.6: Discovery Setup template. New nodes are initialised in *Init*. The *Setup-ChooseDiscoverNode* loop sets which nodes from the existing constellation, that discovers which new nodes. The node that starts the Discovery algorithm is chosen in the edge going out of *ChooseStart*. This setup creates scenarios as seen in Figure 6.8a and Figure 6.8b.

The setup templates for the Discovery model can be seen in Figure 6.6 and Figure 6.7. These setup templates are dependent on two variables, the number of nodes in the existing constellation, and the number of new nodes. The setup templates initialise the new nodes and choose which nodes from the existing constellation has discovered each new node. Lastly, they choose a starting node among the existing nodes that have discovered a new node. The scenarios that are created can be seen in Figure 6.8. The setup template in Figure 6.7, creates a broken link between two nodes.

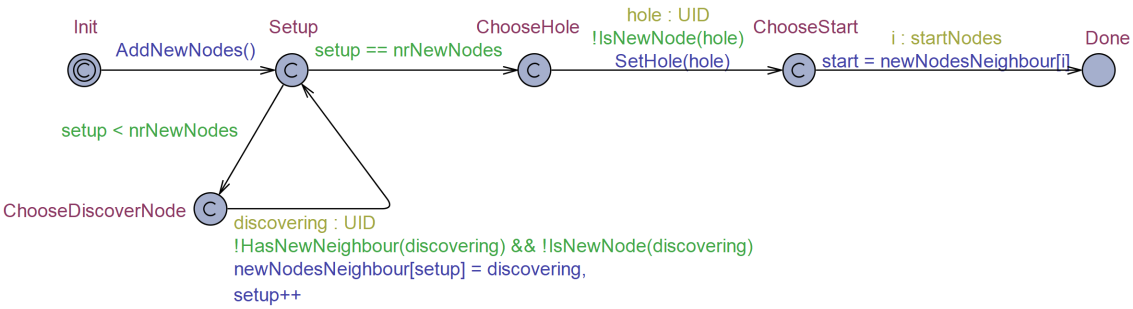


Figure 6.7: Discovery Setup template, with a broken link. Additionally to the other setup template, this creates a broken link between between two nodes in the existing ring. This setup creates scenarios as seen in Figure 6.8c.

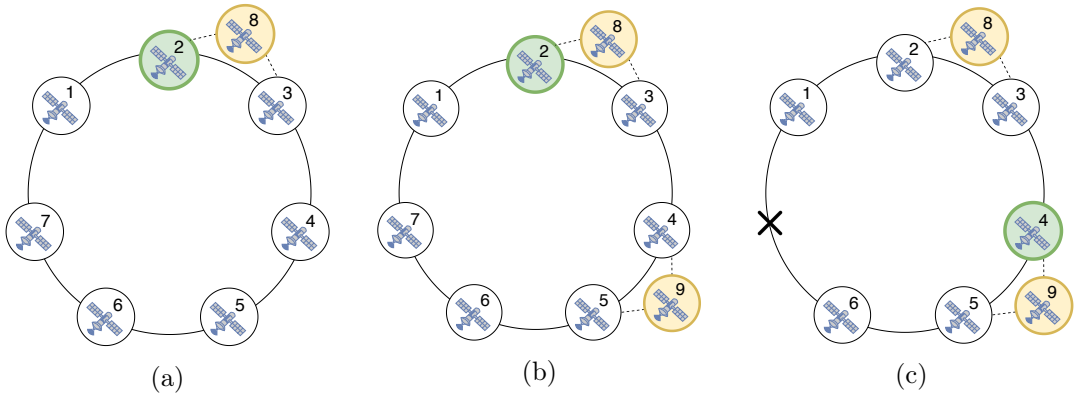


Figure 6.8: Discovery Scenarios. In *(a)*, there is one new node, node 8, in which node 2 discovers it and starts Discovery. In *(b)*, there is two new nodes, node 8 and 9, in which node 2 starts Discovery. In (c) there is no link between node 1 and 6.

Like the Plan Generation model, the Discovery model does not introduce changes between the synchronous and asynchronous node templates. We instead focus on the difference between the node template that only handles a ring with no broken links, as in Figure 6.8a and Figure 6.8b, and the node template that handles all three scenarios described in Figure 6.8. The two different node templates can be seen in Figure 6.9 and Figure 6.10. For both node templates, the asynchronous version does not introduce needed changes, and we leave them out. In the node template that only handles a complete ring, Figure 6.9, nodes only transmit edges that was previously unknown. The first node that does not receive any unknown edges, stops transmitting further, as seen in the `CheckNewKnowledge` location.

The node template that also handles broken links, Figure 6.10, rely on the variables *first pass* and *second pass* to be able to handle a broken link. Nodes that only have one neighbour in the existing broken ring, updates these two variables. To synchronise the network map of all nodes in the broken ring, nodes needs to transmit from one node with one neighbour, to the other node with one neighbour, and back again. Additionally, if either of the *first pass* or *second pass* flag is raised, nodes transmit any knew knowledge, not only knowledge to the specific node. This correctly handles transmitting knowledge, when communication reverses direction. When one of these nodes receive a transmission with the *second pass* flag raised, it can stop. However if neither of the *first pass* or *second pass* flags are raised, in the scenario of a complete ring, nodes communicate like the template in Figure 6.9.

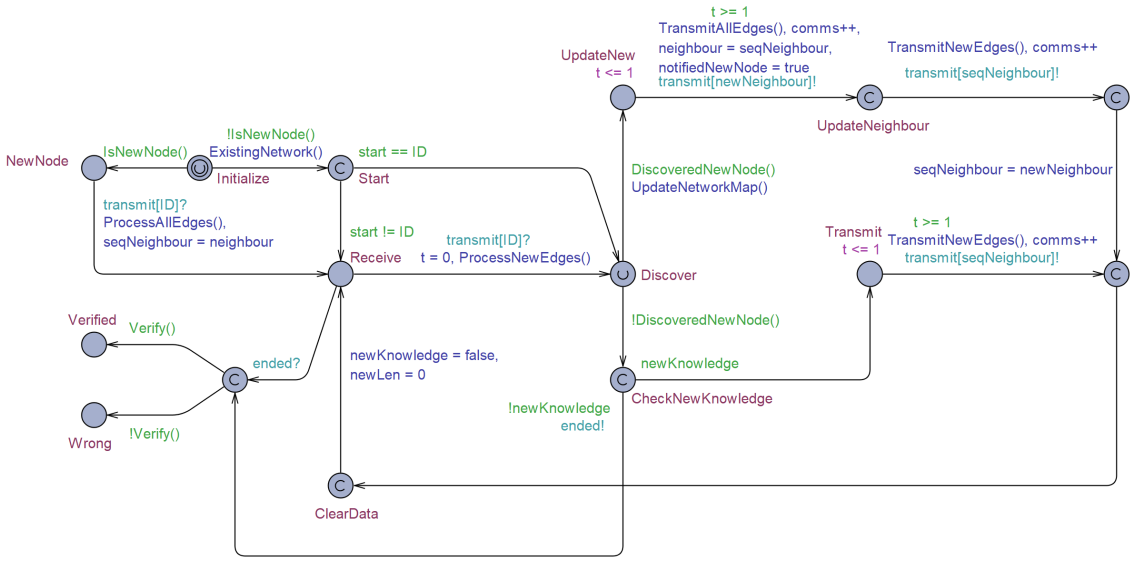


Figure 6.9: Synchronous Discovery node template. In *Initialize*, nodes initialise the existing network map or if the node is one of the new nodes, it goes to the `NewNode` and waits for a transmission. After nodes has received a transmission and processed it, from the `Receive` location, nodes first check if it has discovered a new node, if it has it sends all of its knowledge to the new node. If no new node is discovered, nodes checks if it has new knowledge, and if it has it transmits it to its neighbour, otherwise it goes back to the `Receive` location, ending this round of Discovery.

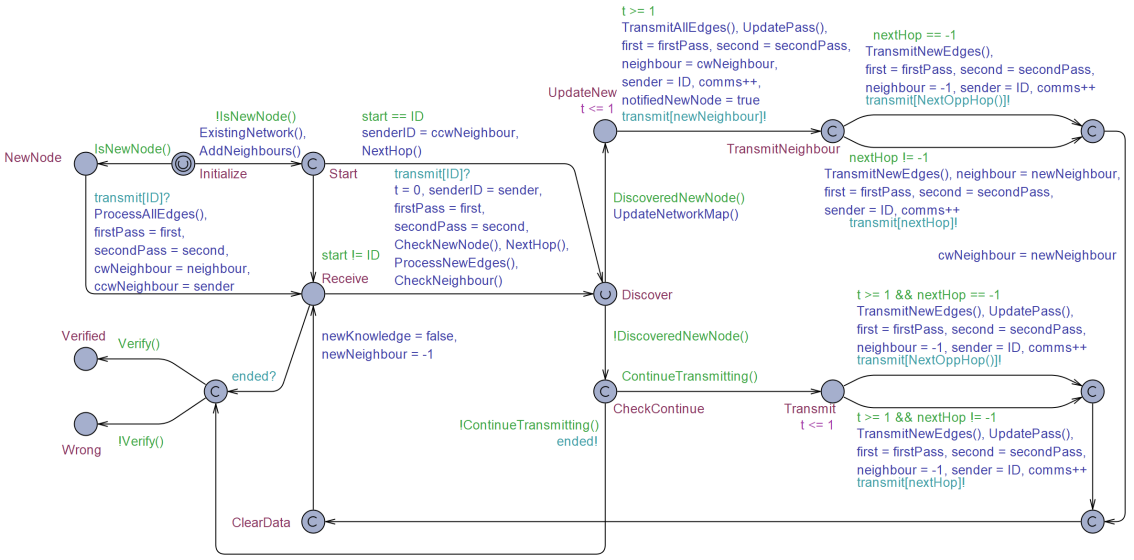


Figure 6.10: Synchronous Discovery node template that handles broken link. Here the template has to handle communication in both directions, clock wise and counter clockwise, and updating nodes neighbours correctly based on new nodes. The main difference, compared to Figure 6.9 is seen in the *Update* statements, in which we have send and use more information to handle the scenario with the broken link.

To verify the correctness, communication complexity, and that all nodes terminate in the Discovery model, we use the queries seen in Listing 6.2. In this model, correctness

is calcaluated in each node template, as each node has to check if the network map it produces is correct. The correctness of the algorithm is then verified by checking that all nodes reach the *Verified* location. Line 2-3 of Listing 6.2 verifies that all nodes eventually terminate. Here, termination is that all nodes are in the *Receive* location when a node stops transmitting Discovery. Line 5-6 verify that all nodes in all traces eventually reach the *Verified* location, and as a result, that the algorithm is correct. Line 8 verify that the upper bound of the number of transmissions to be $(n \cdot 2) - 2$, for a given n, in the scenario with a complete ring. Line 10 verify that the upper bound of the number of transmissions to be at least $(n-1) \cdot 3$, for a given n, in the scenario with a broken link. For both queries, the lowerbound is dependent on which nodes start, which nodes has discovered new nodes, and where the broken link is.

```
1  /* Termination Queries */
2  A<> forall (i : UID) RingNodeSync(i).Wrong || RingNodeSync(i).Verified
3  A<> forall (i : UID) RingNodeAsync(i).Wrong || RingNodeAsync(i).Verified
4
5  /* Correctness queries */
6  A<> forall (i : UID) RingNodeSync(i).Verified
7  A<> forall (i : UID) RingNodeAsync(i).Verified
8
9  /* Communication complexity - Complete Ring */
10 A[] comms <= nodes * 2 - 2
11
12 /* Communication complexity - */
13 A[] comms <= (nodes-1) * 3
```

Listing 6.2: Discovery Queries

In this model we have left out details of how nodes discover new nodes, which is described in Section 4.5. This is left out as it is an individual task and we focus on the distributed aspects of the algorithms in our models. In the model we try to answer, that if new nodes are introduces, will the discovery algorithm correctly share and update each node's network map.

6.3 Failure Detection Model

The Failure Detection model consists of one setup template and, like the previous models, a synchronous and asynchronous version of the node template. The setup template creates three different scenarios, described in Figure 6.12. The node templates models nodes that are running the Failure Detection algorithm. Here, a node cannot reach one of its neighbours and transmits a message around the ring to confirm if other nodes also cannot reach the node.

The setup template for the Failure Detection model can be seen in Figure 6.11. It creates the three different scenarios seen in Figure 6.12 and all their variations in terms of which node is the starting node and which nodes are dead. As a result, when we verify if the algorithm works, we verify that it works for all three scenarios and their variations. The first dead node will always be one of the nodes next to the starting node, to emmulate the starting event of Failure Detection. The starting node then starts transmitting in the

opposite direction in the ring compared to the dead node, in order to start the failure detection algorithm.

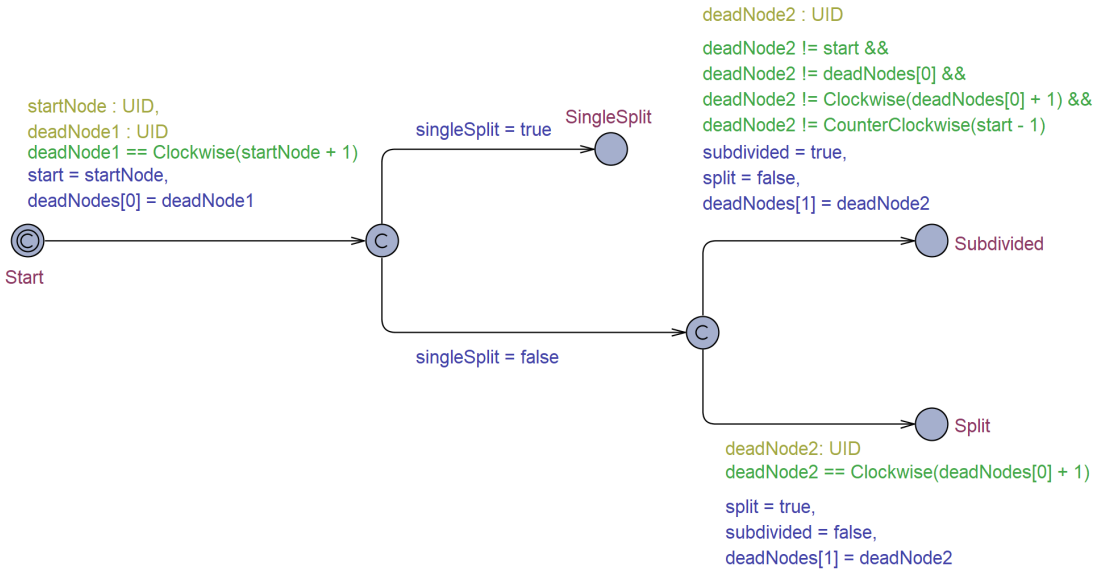
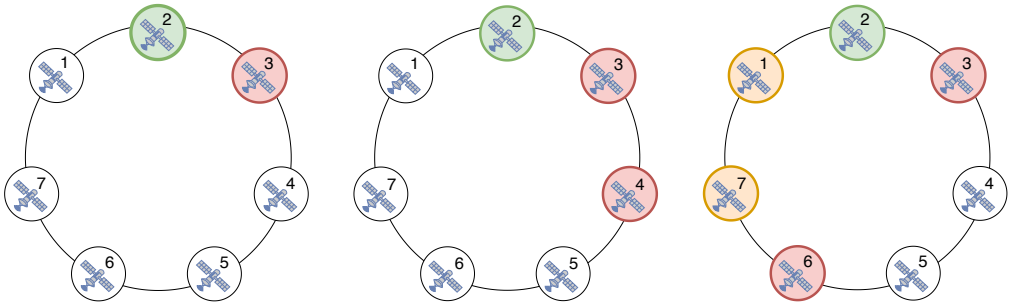


Figure 6.11: Failure Detection setup template. In *Start* a starting node is selected and its clockwise neighbour is set as dead. From here there is three possible scenarios to setup: *SingleSplit* with only one dead node, *Split* an additionally dead node, that is a neighbour to the first dead node, and *Subdivided* with an additionally dead node that is not a neighbour to the first, resulting in a subdivision of the ring.



(a) scenario 1 (b) scenario 2 (c) scenario 3

Figure 6.12: Failure Detection scenarios. In all scenarios, node 2 starts and sends a Failure Detection transmission in the clockwise direction, as it has not received any response from node 3. In scenario *(a)*, node 3 is dead which will be confirmed by node 4. In scenario *(b)*, node 3 and 4 are dead, in which node 5 discovers that node 4 is dead. In scenario *(c)*, node 3 and 6 are dead, in which node 7 discovers that node 6 is dead, and that the constellation is now subdivided.

The synchronous and asynchronous node templates can be seen in Figure 6.13 and Figure 6.14 respectively. For nodes to be able to detect that another node is dead, nodes expect a response or confirmation when communicating. Additionally, rules for when a node is considered dead is necessary. In this model, a node is considered dead when it fails to respond three times in a row. This can be seen in the *Transmit-Response-*

Retry-Transmit-loop in both templates. In the asynchronous node, this loop has to be handled differently than the synchronous. This can be seen in the extra outgoing edge from *Transmit*. Here nodes has to expect a response while it is ready to transmit, after it has transmitted once, as the node cannot expect a response immediately after transmitting, as the synchronous node can.

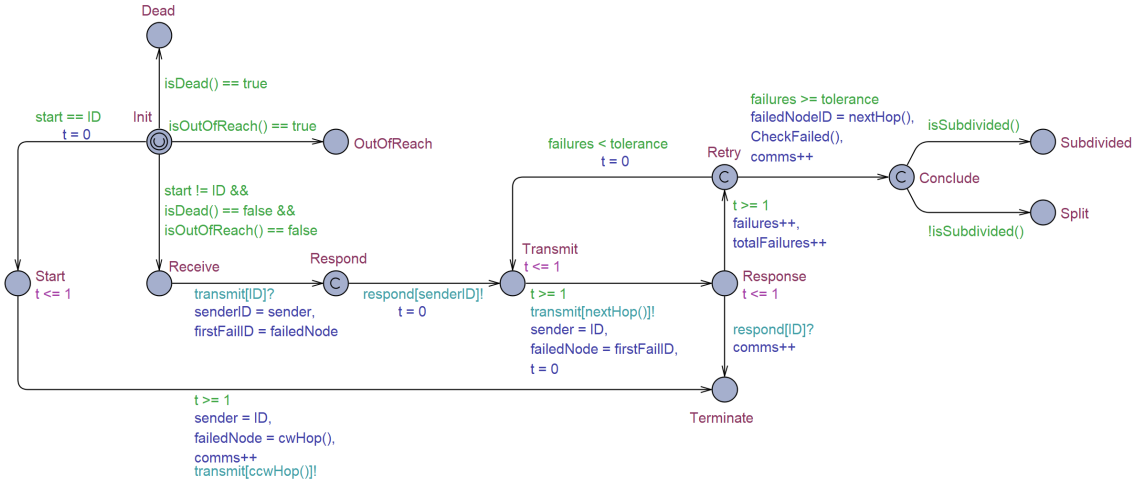


Figure 6.13: Synchronous Failure Detection node. All nodes communicate to the next node in the ring until one node does not receive a response. When a node has received a Failure Detection transmission and discovers a node that does not respond, it compares the transmitted dead node to the one it just discovered. Here, it concludes whether the the constellation is subdivided or not, as seen in the *Conclude* location.

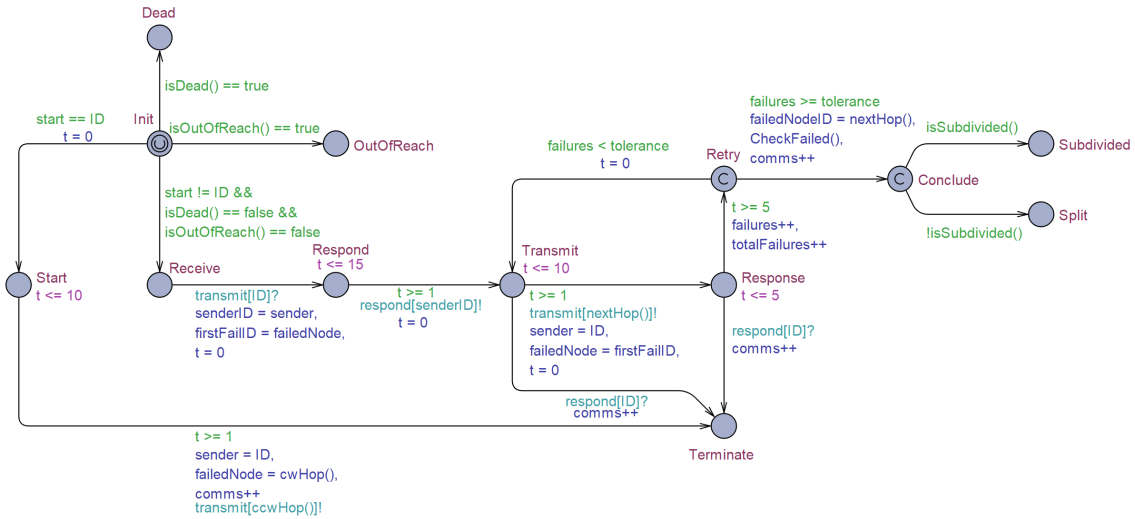


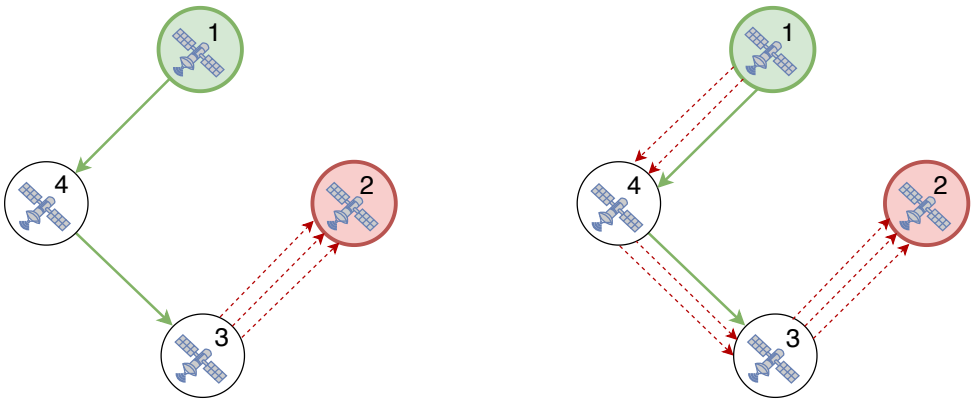
Figure 6.14: Asynchronous Failure Detection node. Compared to the synchronous version, transmit-, response- and wait time are arbitrary between 1 and some upper bound. The upper bound in *Respond* ensures that nodes responds before being considered dead. Each node also have to expect a response while time passes before transmitting.

Verification of the Failure Detection model has to consider that nodes may be in the *Dead* or *OutOfReach* locations and never participate in the communication. Additionally, only

one node reaches the *Split* or *Subdivided* locations, other active nodes reach the *Terminate* location. This results in five different ending locations for nodes. Correctness in this model, is that one node correctly outputs whether the ring is split or subdivided.

```
1  /* Termination Queries */
2  A<> forall (i: UID) RingNodeSync(i).Dead || RingNodeSync(i).OutOfReach || RingNodeSync(i).Split ||
3                      RingNodeSync(i).Subdivided || RingNodeSync(i).Terminate
4  A<> forall (i: UID) RingNodeAsync(i).Dead || RingNodeAsync(i).OutOfReach || RingNodeAsync(i).Split ||
5                      RingNodeAsync(i).Subdivided || RingNodeAsync(i).Terminate
6
7  /* Correctness Queries */
8  FailureDetection.Subdivided  -->  (exists (i : UID) RingNodeSync(i).Subdivided)
9  FailureDetection.Split       -->  (exists (i : UID) RingNodeSync(i).Split)
10 FailureDetection.SingleSplit -->  (exists (i : UID) RingNodeSync(i).Split)
11
12 /* Communication Complexity */
13 A[] comms <= N - 1
14
15 /* Sync Failures */
16 A[] totalFailures <= 3
17
18 /* Async Failures */
19 A[] totalFailures <= ((N - 3) * 2) + 3
```

Listing 6.3: Failure Detection Queries



(a) Synchronous (b) Asynchronous

Figure 6.15: Number of failures in Failure Detection. In the synchronous model, *(a)*, each node responds immediately. As a result, failed communication (dashed arrows) only happen at the the dead node, node 2, where its neighbour, node 3, tries three times before concluding it dead. In the asynchronous model, *(b)*, each node responds after arbitrary time with an upperbound so that it always responds before it would be concluded dead. As a result, each communication may be seen as failed two times, and three times with the dead node, node 2.

The queries for the Failure Detection model can be seen in Listing 6.3. The queries that check that all nodes ends in one of the five ending locations can be seen in line 2 and 4. The queries that verify correctness can be seen in line 6-8. Here, the queries check that one node, in each scenario, reach the correct end location. The communication upper bound is verified in line 13. The communication complexity is dependent on the number of links between the start node and the first encountered dead node. The upper bound reflects the scenario, in which there is only one dead node, as the nodes has to communicate all the way around the ring to reach the neighbour to the dead node. Line 16 and 17 check the upper

bound of the number of failures. The expected number of failures in the synchronous environment is only 3, as it is only communication to the dead node that fails. In the asynchronous environment, every time two nodes communicate there is a chance that the transmitting node tries communicating two times before receiving a response. It is only when encountering the dead node that it also fails a third time. The difference between the two environments is seen in Figure 6.15. Like the communication queries, the number of failures is dependent on which node are the start and first encountered dead node.

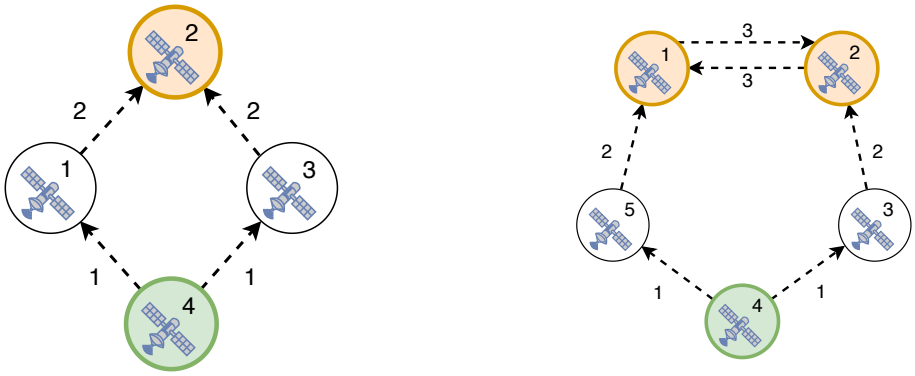
6.4 UPPAAL Experiments

As our models only have been sending transmissions in one direction at a time, we explore what it takes to convert one of our models to utilise converge casting, in which a node can send transmissions to both its neighbours at the same time. We also test verification capabilities, in terms of how long it takes to verify correctness, and how verification time increases as the number of nodes increases. Lastly, we use UPPAAL SMC to compare different algorithms used in the Plan Generation model.

6.4.1 Bidirectional Discovery

As the discussed models and algorithms, Plan Generation, Discover, and Failure Detection, communicate in one direction, we wanted to explore how bidirectional algorithms would work and how it would be modelled. To explore this, we have modelled and implemented the Discovery algorithm, that is now based on a previous version of the algorithm, with bidirectional communication. In the previous version, each node started only with knowledge of their two neighbouring nodes, in which the previous Discovery Algorithm would take exactly two passes around the ring, to gather complete knowledge in the first pass and passing it to all nodes in the second pass. Although it is not something we have focused on in this project, the benefit of utilising bidirectional communication, is an improvement in communication time. A bidirectional version of the discovery algorithm, would use half the time of the unidirectional version, as the ring is being traversed in parallel.

As the previous version of the Discovery algorithm, in the case of a split constellation, it takes two passes to complete, it also requires two converge casts to complete the bidirectional version. One converge cast to start the communication in both directions and a second converge cast to communicate the opposite direction when the first cast converges on a node. Here a converge cast is understood as a node communicating with both its neighbour at the same time.



(a) Normal converge

(b) Synchronised converge

Figure 6.16: Converging transmissions in a synchronised environment. In both figures, node 4 starts with converge casting to its two neighbours. The number on the edges represents the round in which the transmission happens. *(a)* shows that with an even number of nodes, one node will receive transmissions from two nodes in the same round, and be able to detect converged transmissions. *(b)* shows that with an uneven number of nodes, node 1 and 2, will communicate with each other in the same round. Here, one node cannot decide to be the converging node without extra logic.

Distributed algorithms that have bidirectional links have to handle when communication converges. In our case, with a ring of nodes, there are two types of converging communication, as seen in Figure 6.16. In Figure 6.16b, one node has to decide that it is the one who starts the next converge cast. If both were to start a converge cast, it would result in double communication on the way back to the starting node. In a synchronous environment, as seen in Figure 6.16, we can predict what scenario we are in and what satellites would converge. In an asynchronous environment we cannot predict these two things, and the algorithm should be able to handle both.

The synchronous node template that implements the bidirectional version of the previous Discovery algorithm can be seen in Figure 6.17. A node detects a normal converge if it receives two transmissions in the same time unit. This can be seen in the edge between *Wait* and *SingleConverge*. Otherwise, if a node receives the second transmission in the same time unit in which it would normally have transmitted to its next neighbour, it detects a synchronised converge. To ensure correct behaviour, as seen in Figure 6.16b, the node first sends a transmission to the sending node, resulting in both nodes reaching the *SyncConverge* location. This is seen in the *Transmit* and *ResetOrEnd* locations to the *SyncConverge* location. To decide which node should start the next converge cast, each node checks if its *ID* is greater than the sending node's *ID*. If it is greater, the node decides to be the node that starts the next converge cast.

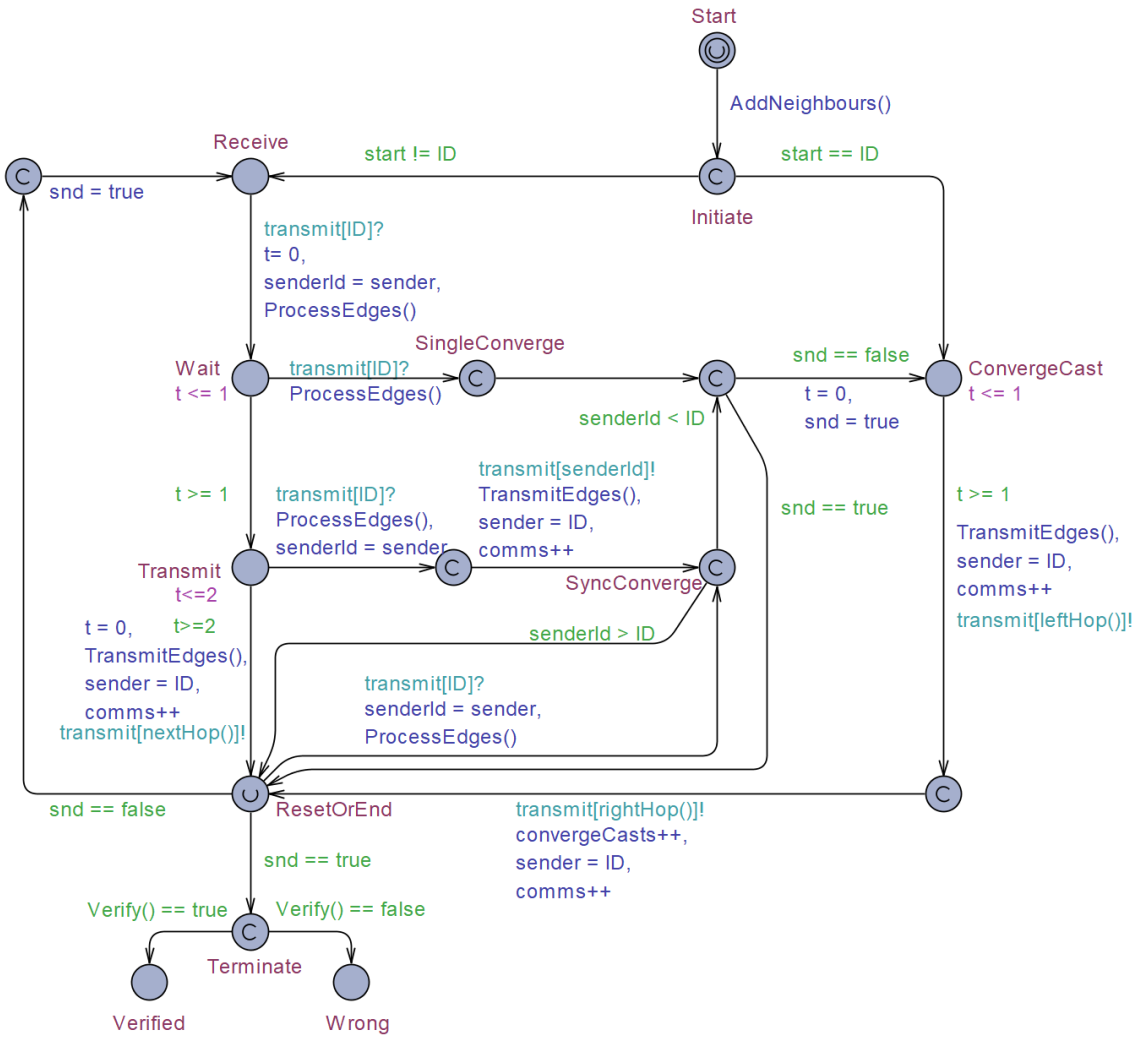


Figure 6.17: Synchronous bidirectional Discovery node template. Here, the starting node converge casts, as seen in the *ConvergeCast* location. Nodes receive messages and transmit them to their following neighbour unless it detects converging communication. This is detected in the *SingleConverge* and *SyncConverge*, which detects the two scenarios in Figure 6.16 respectively. In the *SyncConverge* location, both nodes that have crossed transmissions, will check the *ID* of the sender and decide whether or not to start the next converge cast.

Modelling correct behaviour of converging messages proved to be a challenge in UPPAAL, especially in the scenario in which two nodes communicate to each other in the same round. Notice how the model uses two time steps in the *Wait* and *Transmit* location, and expects a new transmission in both the *Transmit* and *ResetOrEnd* locations. This is done to accurately model the converging communication, but results in a difference between time units and rounds, in which the time units after completing is higher than the number of rounds used.

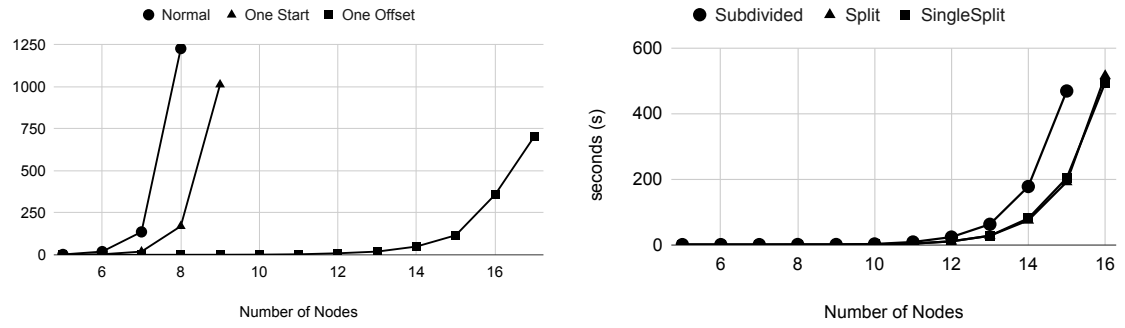
```
1  /* Correctness Queries */
2  A<> forall (i : UID) RingNodeSync(i).Verified
3
4  /* Uneven Number of Nodes */
5  A[] comms <= (N * 2) + 2
6  E<> forall (i : UID) RingNodeSync(i).Verified  && comms <= (N * 2) + 1
7
8  /* Even Number of Nodes */
9  A[] comms <= N * 2
10 E<> forall (i : UID) RingNodeSync(i).Verified  && comms <= (N * 2) - 1
```

Listing 6.4: Bidirectional Discovery Queries

The queries we use to verify the bidirectional version of the previous Discovery model can be seen in Listing 6.4. Similar to the Discovery queries, we verify that each node has the same and correct network map after the algorithm terminates, as seen in line 1. In line 5 and 6 we verify that the algorithm uses exactly $n \cdot 2 + 2$ transmissions for a given uneven n. Note that this is two extra transmissions than the unidirectional version. These extra two transmissions happen due to the synchronised converges. Line 9 and 10 verify that the algorithm uses exactly $n \cdot 2$ for a given even n. This shows that there is a small overhead for utilising bidirectional communication in our Discovery algorithm. However, the overhead does not increase as the number of nodes increases, as there will always be a maximum of two synchronised converges.

6.4.2 Verification Time

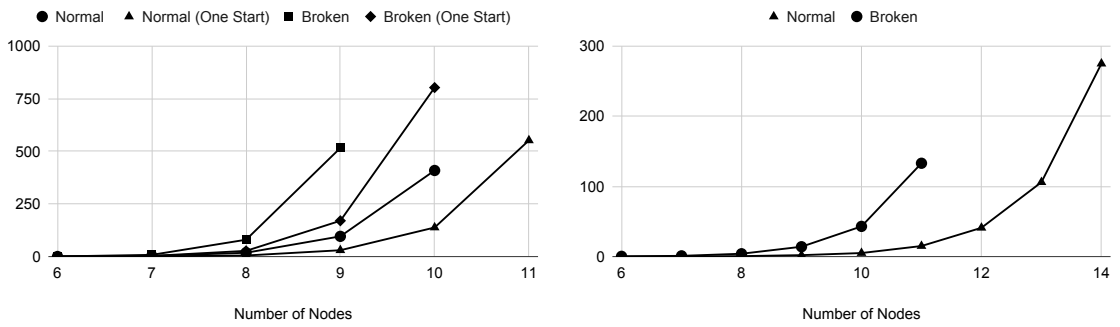
To explore the verification capabilities of our models, we test the verification time to verify correctness in our models. Here, we are interested in exploring how verification time scales with the number of nodes in the network. All verification tests ran with the same parameters and the same search order in UPPAAL, in which we used *Random Depth First*.



(a) Plan Generation

(b) Failure Detection

Figure 6.18: Verification time for verifying correctness in the Plan Generation and Failure Detection model. For Plan Generation, *(a)*, we compare the verification time to the normal setup, and setups that reduce the number of scenarios, namely: The same starting node and the same offset position. For Failure Detection, *(b)* we test the verification time for the three different scenarios. Note here, that to test complete correctness, all three scenarios have to be correct.



(a) Discovery - Three new nodes

(b) Discovery - One new node

Figure 6.19: Verification time for verifying correctness in the Discovery model. Here, we test both the scenarios, a complete ring and a ring with a broken link. In *(a)* we test the verification time compared to the number of nodes for both scenarios, with three new nodes. It also compares the verification time to the case in which there is only one starting node. In *(b)* we also test the verification time when there is only one new node, to compare to the result from three new nodes.

Figure 6.18a shows the verification time for the Plan Generation model. It shows that verifying all the different scenarios, especially the various scenarios with different offsets of target positions, quickly scales to long verification times. Figure 6.18b shows the verification time of the three different correctness queries for the Failure Detection model. It shows that verification time for the subdivided scenario takes longer time and in all scenarios, verification time quickly scales after 14 nodes. The longer verification time in the subdivided scenario is likely due to the selection of the second dead satellite, compared to the *Split* scenario, in which once the first dead node is chosen, the second dead node has to be next to the first.

Figure 6.19 shows the verification time for the Discovery mode. Here, we see that the scenario in which there is a broken link takes more time to verify. Comparing Figure 6.19a and Figure 6.19b we see that verifying more than one new node takes more time. This is likely due to the increased number of scenarios, as new nodes are selected in the model, and the model has to verify each possible selection.

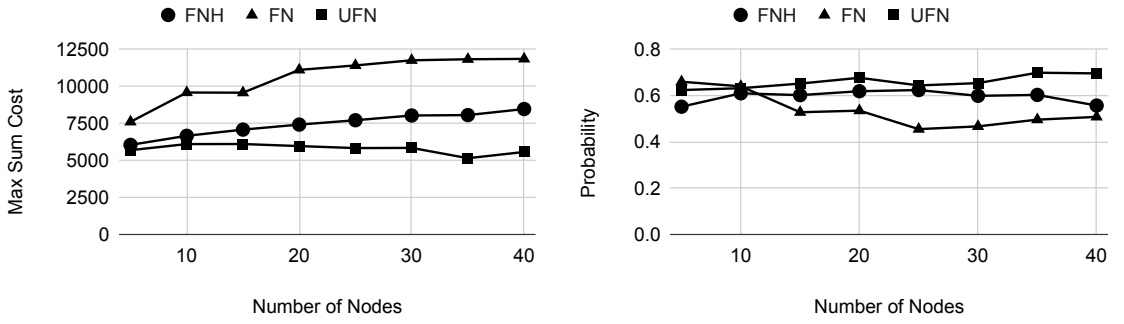
6.4.3 UPPAAL SMC - Plan Generation Algorithms

We have explored alternative algorithms for choosing target positions in the Plan Generation algorithm. Here, we use UPPAAL SMC to explore how the different algorithms affect the combined cost of all nodes. We test three different algorithms, a greedy algorithm, in which nodes take the best target position without considering any other node, an algorithm that avoids taking target positions that are its two neighbours' best, and the implemented algorithm, that considers all nodes. We have nicknamed them *Unfriendly Neighbour*, *Friendly Neighbour*, and *Friendly Neighbourhood* respectively.

```
1  /* Max sum */
2  E[ <= N*2; 10000](max: sum(i: UID) RingNodeSync(i).cost)
3
4  /* Probabilty of sum being less than max sum */
5  Pr[ <= N*2](<> (PlanGeneration.Correct && (sum(i: UID) RingNodeSync(i).cost) < MAX_SUM))
6  Pr[ <= N*2](<> (PlanGenerationHole.Correct && (sum(i: UID) RingNodeSync(i).cost) < MAX_SUM))
7
8  /* Visualise sum */
9  simulate [<=N*2; 100] {sum (i : UID) RingNodeSync(i).cost}
```

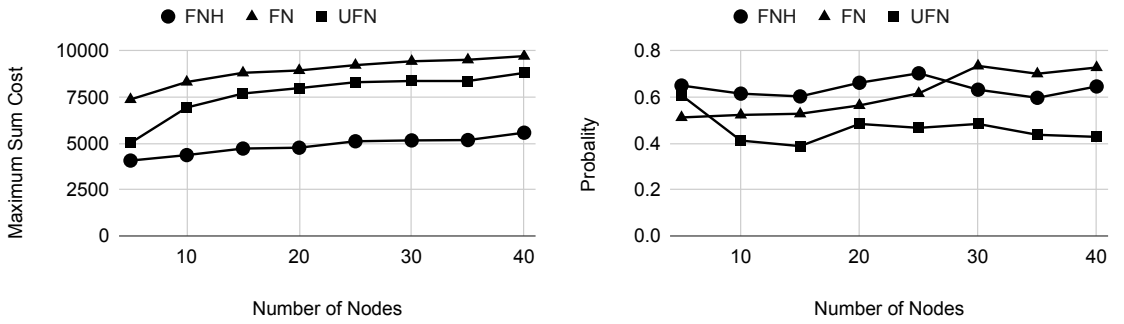
Listing 6.5: Bidirectional Discovery Queries

The UPPAAL SMC queries can be seen in Listing 6.5. The query in line 2 finds the average maximum sum of all nodes' cost in 10.000 runs and within $n \cdot 2$ time units. Note that the query tests the synchronised nodes, which always takes exactly $n \cdot 2 - 1$ time units to complete. Line 5 and 6 find the probability of the combined sum being less than a specified value, after the nodes have terminated. Here, we supply the value from the previous query to the value tested in these queries. We use these two queries to test how often the different algorithms produce a combined sum less than the average maximum sum. Line 9 shows the query we use to visualise how the sum increases over time, in 100 runs. We use this query to explore patterns of how the combined cost increases.



(a) Average Maximum Cost Sum (b) Probability of Sum Cost being less than Max

Figure 6.20: Plan Generation: evenly-spaced-scenario

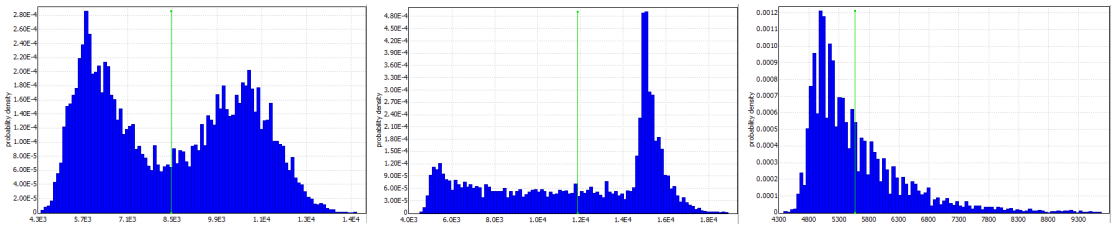


(a) Average Maximum Cost Sum (b) Probability of Sum Cost being less than Max

Figure 6.21: Plan Generation: hole-scenario

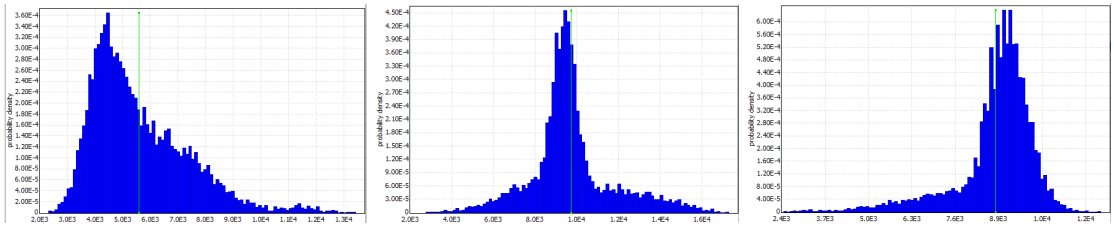
Figure 6.20 and Figure 6.21 show the average maximum cost sum and the probability of the sum being less than the max sum, as the number of nodes increases. The figures

show the results from the two different scenarios, evenly distributed nodes and distributed nodes with a hole, see Section 6.1. The *Friendly Neighbourhood* algorithm is significantly better in the hole-scenario, as the maximum sum cost is lower while the probability of the sum being less than max is better or similar to the other algorithms. The *Unfriendly Neighbour* algorithm seems to produce the best results in the evenly-spaced scenario, but not in the hole-scenario. Lastly the *Friendly Neighbour* algorithms produce worse results in the evenly-spaced scenario.



(a) Friendly Neighbourhood (b) Friendly Neighbour (c) Unfriendly Neighbour

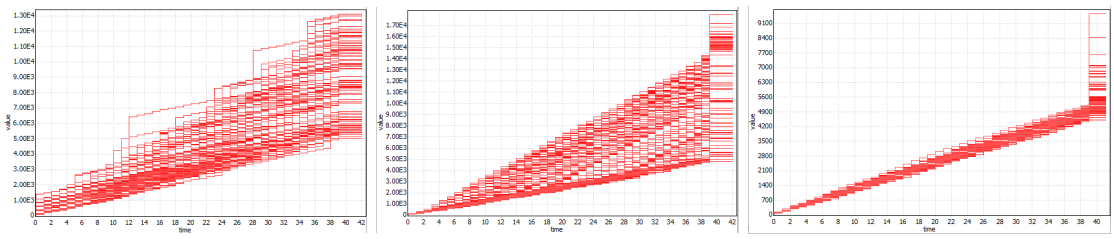
Figure 6.22: Plan Generation evenly-spaced-scenario: max cost sum distribution for 40 nodes.



(a) Friendly Neighbourhood (b) Friendly Neighbour (c) Unfriendly Neighbour

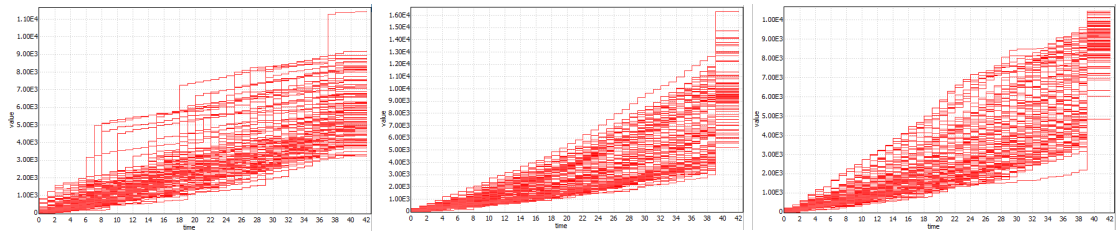
Figure 6.23: Plan Generation hole-scenario: max cost sum distribution for 40 nodes.

UPPAAL SMC also creates a distribution graph of the maximum cost sum from the query in line 2 of Listing 6.5. To further explore the behaviour of our algorithms, Figure 6.22 and Figure 6.23 show the distribution of maximum cost sum in the specified 10.000 runs, with 40 nodes, in the two different scenarios. Here we see that the *Unfriendly Neighbour* algorithm has the opposite distribution in the evenly-spaced and hole scenario, which shows that it is likely to produce a worse result than e.g. the *Friendly Neighbourhood* algorithm in hole scenario. This is also supported by the low probability of the sum being less than the maximum sum cost, seen in Figure 6.21a.



(a) Friendly Neighbourhood (b) Friendly Neighbour (c) Unfriendly Neighbour

Figure 6.24: Plan Generation evenly-spaced-scenario sum cost over 100 runs



(a) Friendly Neighbourhood (b) Friendly Neighbour (c) Unfriendly Neighbour

Figure 6.25: Plan Generation hole-scenario sum cost over 100 runs

Lastly, we use the *simulate* query, see line 9 in Listing 6.5, to visualise how the combined cost grows as each node choose a target position. Figure 6.24 and Figure 6.25 show the results from this query for the three different algorithms, and in the two different scenarios, respectively. Here we see that the *Unfriendly Neighbour* algorithm in both scenarios often results in the last being left with a target position that is far away from the node. The *Friendly Neighbour* algorithms seem to have the similar behaviour, but also produce more spread results in the evenly-spaced scenario. The *Friendly Neighbourhood* avoids the last node having a high cost choice, but it seems like there is at least one node that has a high cost, as seen in the many high jumps in cost, especially in Figure 6.25a.

7 Conclusion

In this project we have created distributed algorithms for introducing more autonomy in satellite constellations or formations. Our algorithms handle failed links or dead satellites and are able to recover to a functional constellation. The algorithms also handle discovery of new satellites to the constellation, in which the network topology is synchronised in the constellation. With these algorithms, the need for manually maintaining satellite constellations or formations is reduced, as problems can be discovered and recovered from by the satellites autonomously.

The project consists of three major contributions, a satellite constellation framework, a visual simulation application for said framework, and design and verification of UPPAAL models based on operations from the framework.

The primary focus of the framework is to allow satellites to autonomously handle stopping failures of satellites, allowing it to maintain a continuous communication ring. In order to do this, the framework has the capability to detect failed, moved, or added satellites as well as reorganisation of a satellite constellation in order to close link holes. The framework can be run in the simulator described below.

The simulation tool allows one to trigger operations in a satellite constellation and view how the operations work as they are performed. New satellites can be added to the constellation and satellites can be toggled on/off as to create holes in the ring network. One equatorial ring of satellites is visualised from a top-down view as well as messages communicated between satellites and locations that are to be filled in a new constellation.

UPPAAL has been used to model the plan generation, discovery, and fault handling operations as well as verify properties and analyse performance. We verify termination, correctness, and communication complexity for the three operations. Additionally, we have used UPPAAL SMC to analyse the performance of the plan generation algorithms and variations of it.

Bibliography

[1] B. Bresson, J. B. Johansen, and K. H. Røgen, *Modelling and Simulating Challenges in Satellite Constellations*. https://projekter.aau.dk/projekter/files/319759161/DS907e19_Modelling_and_Simulating_Challenges_in_Satellite_Constellations.pdf, 2020. Visited: 11/05-2020.

[2] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal smc tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, pp. 397–415, Aug 2015.

[3] https://people.cs.aau.dk/~adavid/cora/. Visited: 12/05/2020.

[4] http://people.cs.aau.dk/~adavid/tiga/. Visited: 12/05/2020.

[5] https://people.cs.aau.dk/~marius/stratego/. Visited: 12/05/2020.

[6] G. Behrmann, A. David, and K. G. Larsen, *A Tutorial on Uppaal*, pp. 200–236. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[7] M. Rabbat and R. Nowak, "Distributed optimization in sensor networks," pp. 20–27, 04 2004.

[8] G. Mushet, G. Mingotti, C. Colombo, and C. Mcinnes, "Self-organising satellite constellation in geostationary earth orbit," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 51, no. 2, pp. 910–923, 2015.

[9] D. O'hallaron, B. Lowekamp, and T. Gross, "Topology discovery for large ethernet networks," *ACM SIGCOMM Computer Communication Review*, vol. 31, 07 2001.

[10] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[11] B. R. Heap, "Permutations by Interchanges," *The Computer Journal*, vol. 6, pp. 293–298, 11 1963.