Bachelor project
A Peer-to-Peer DCR graph engine

Frederik Tollstorff de Voss {lvw655}
Andreas Nienstædt Halling Larsen {pjk293}

July 7, 2020

Contents

1 Abstract

This paper addresses the difficulty of effectively executing collaborative declarative processes using a distributed system.
While others have taken the approach of using imperative process modelling notations and a centralised actor in order to model business processes, we argue that the declarative process modelling notation is better suited to accurately represent the complexities often occurring in knowledge-intensive, real life processes, and that the process should instead be distributed across several actors. To do this, we present a design and implementation of a peer-to-peer system that utilises the declarative process modelling notation known as Dynamic Condition Response (DCR) graphs, which also functions as execution model. We achieve a safe distribution and execution of DCR Graphs across actors by using our message protocol, which uses message propagation to propagate changes in state to relevant actors, as well as a blocking mechanism that ensures that all actors part of the global DCR graphs are synchronised, and therefore do not maintain a non-conforming state.
Our implementation is made using the object oriented programming language C#, and the .NET Socket library, and is tested using unit- and integration testing. Challenges were faced in the networking aspect of the design, as there are many scenarios where the effects of a failure in one part of the distributed system affects other parts of the system, e.g. due to messages being unable to be propagated by the failed part. We conclude that with our design, executing DCR graphs in a peer-to-peer network is possible by utilising different mechanisms to ensure the synchronicity of the global graph.

2 Introduction

Currently, the most commonly used way to model business processes is by describing the steps that make up a given process from start to end, *imperatively.* In these imperative process modelling notations, nothing is allowed by default, and it is up to the designers themselves to structure the exact flow of how these steps should be taken, and make exceptions for edge cases where special rules apply. This imperative way of modelling processes is very efficient for modelling rigid processes with little variation, such as a factory assembly line or a cooking recipe, but it falls short when it comes to the opposite, more flexible processes, where there are fewer restrictions and more ways in which that process can be completed. An example of this, is shopping for groceries, where items on the grocery list may be put in the shopping cart in any order.

Instead of modelling such a process imperatively, it would be more efficient to model it *declaratively.* In declarative process modelling, everything is allowed by default, and constraints are then defined in order to model forbidden behaviour. In declarative process modelling, the model describes the constraints of the process, whereas in imperative modelling the flow of the process is described. As we see in [4][5] the lack of flexibility in the imperative model is highlighted when the structure of the process it is trying to model changes, i.e the factory assembly line changes production or the as-

sembly line itself changes. Since the imperative model captures how a process is preformed, changes to that process make the model difficult to change without having to rework a sizeable portion of the model or sometimes fully reworking the model. Handling new constraints in the declarative model is easier. Since it is focused on modelling the constraints, the new constraint is simply added to the model without having to do a larger rework of the model.
One such declarative process model is Dynamic Condition Response graphs, abbreviated DCR graphs, which has already been used successfully in order to capture business processes [5]. While we believe DCR graphs are well suited to model these business processes, past implementations of DCR graphs have consisted of a global graph in a centralised system. Real life business processes may often involve many different actors, located in different parts of the world, and do not necessarily have a primary governing actor. Drawing inspiration from [2], we have created and implemented our own design for distributing different parts of the global DCR graph across different actors by using a peer-to-peer system. Peer-to-peer systems offer a way to connect to others via the internet without the need to query a server first, by letting peers connect directly to each other. This type of architecture is useful for information sharing and file sharing, as seen with Napster. In [6] we see a definition of a peer-to-peer system as when participants of the network share some of their hardware with others, i.e files that are stored on their local computer.

In the same way, a peer-to-peer architecture can be useful when working with DCR graphs, where a process can be split into multiple parts and be distributed. An example of this could be a doctor prescribing medicine to a patient, where part of the process is the doctor prescribing the medicine and the second part is the pharmacy receiving the prescription. This is reflected in reality since the doctor rarely has his doctor's office at the pharmacy and thus, the need for splitting the process into smaller parts arise. It is this need that we attempt to address in this thesis by designing and implementing our own distributed DCR graph engine. In doing so, we will first present formal theory of DCR graphs in section 3 and how they are implemented in a non-distributed setting. We will then present our design and implementation of distibuted DCR graphs in section 4, including the structure of the underlying peer-to-peer network. Finally, we will asses our implementation in section 5, how the system is tested in section 6 and lastly how to run the system in section 7.

3 Dynamic Condition Response graphs

In this section we will provide a general overview of DCR graphs by describing events and the different markings and relations, before giving a formal definition of DCR graphs. Lastly, we showcase our implementation of DCR graphs and relate it to the aforementioned theory.

3.1 DCR Graphs overview

DCR graphs is a declarative workflow notation which can be used to model business processes. These graphs consist of activities, henceforth interchangeably referred to as events, and relations between these events, which serve to create rules for when these events may occur. We henceforth refer to an event occurring as the *execution* of that event. Relations are directed, which means that if an event e_1 has a relation to an event e_2, it does not imply that e_2 has a relation to e_1. Events also have markings, which indicate three different states. These markings are:

- Executed, used to indicate whether an event has been executed before.
- Included, used to indicate whether an event is currently included in the graph. When an event is not included, we say that it is excluded and the event as well as its relations to other events are inactive, meaning the event may not be executed and its relations do nothing. Note that this exclusion does not apply to relations going *to* and not *from* this event.
- Pending, used to indicate whether an event has to be executed at some point in order for the DCR graph to reach an accepting state. When no events are marked as pending, or the pending event(s) are not included, we say that a graph is in an accepting state.

Relations are directed from one event to another, and can have a variety of different effects. As DCR graphs are declarative, events are, by default, executable in any order. It is up to relations to describe the rules for these executions. We include the *milestone* relation, an extension to the preexisting types of relations, introduced in [1]. These relations are:

- Include $\rightarrow+$
  - If $e_1 \rightarrow+ e_2$, then executing e_1 will change the marking of e_2, such that e_2 is included. If e_2 is already included, this has no effect.
- Exclude $\rightarrow\%$,
  - The opposite of included: If $e_1 \rightarrow\% e_2$, then executing e_1 will change the marking of e_2, such that e_2 is no longer included.
- Condition $\rightarrow\bullet$
  - If $e_1 \rightarrow\bullet e_2$, this means that e_1 is a condition for e_2, and e_2 cannot be executed before e_1 is marked as either executed or is not included.

- Response $\bullet\!\!\rightarrow$
  - If $e_1 \bullet\!\!\rightarrow e_2$, then executing e_1 will change the marking of e_2, such that e_2 is pending.
- Milestone $\rightarrow\!\diamond$
  - If $e_1 \rightarrow\!\diamond e_2$, and e_1 is marked as pending, then e_2 cannot be executed. If e_1 is not marked as pending, or is not included, this relation has no effect.

Note, that while DCR graphs may contain cycles, some cycles can result in deadlocks or livelocks.

Definition: Deadlock
A DCR graph is said to be in the state of deadlock if it is not possible to execute any events, before the graph has reached an accepting state - meaning that one or more events are still pending.

Definition: Livelock A DCR graph is said to be in the state of livelock if it is not possible to reach an accepting state, but one or more events are still executable.

An example of a deadlocked DCR graph could be; if we have two events, event e_1 and e_2. If e_1 has a condition relation to e_2 and e_2 has a condition relation to e_1. If one of the two events is marked as pending, then the graph would be in a deadlock since it cannot reach an accepting state. The reason for this is that none of the events can execute because it requires the event at the other end of the condition relation to be executed first.

If we add a third event e_3 to the graph, which is included and has no relations, then the graph would instead be livelocked, since the e_1 and e_2 still cannot execute but e_3 can an indefinite amount of times. In this state, the graph will never become accepting, but e_3 will be able to execute.

3.2 Formal theory of DCR graphs

In this section we will present the formal definition of DCR graphs as we see it in [3]. A DCR graph is a directed, unweighted graph, where vertices are events and the edges are the relations between the events. A DCR graph is defined as such:

3.2.1 Definition 1: A DCR Graph

A DCR graph is defined as a tuple with (E, R, M). Where E is the finite set of labelled events. R is the 5 different types of relations in the graph and M is the markings of the graph. M is defined as a triple (Ex, Re, In). Where Ex is the executed events in the graph. Re is the events that are currently pending in the graph and In is the events which are currently included in the graph.

If G is a DCR graph we have that $E(G)$ is the set of events in G and $Re(G)$ is the set of pending events in G. This also hold for the rest of the markings and relations.

Next, we provide a definition for when a event is enabled and when an event is executed. Before an event can be executed, it must be enabled.

3.2.2 Definition 2: Enabled Events

An event is enabled when it is included and any conditions targeting the event have been resolved, either by the source of the condition being excluded or that the source event has been executed. Lastly, any milestones targeting the event with a unresolved pending marking will also prevent the event from being enabled. If we have a DCR graph $G = (E, R, M)$ with marking $M = (Ex, Re, In)$. We have that $e \in E$ is enabled and write $e \in \text{enabled}(G)$ iff:

$$\text{e} \ \in \ \text{In} \tag{1}$$

$$\text{In} \ \cap (\rightarrow\!\bullet\ e) \subseteq \ \text{Ex} \tag{2}$$

$$\text{In} \ \cap (\rightarrow\!\diamond\ e) \subseteq \ \text{E} \setminus \text{Re} \tag{3}$$

Meaning that, (1) if e is included. (2) if there are no unresolved conditions to e, i.e they have all been executed or excluded. (3) if there are no pending milestones targeting e.

3.2.3 Definition 3: Execution

We have a DCR graph $G = (E, R, M)$ with marking $M = (Ex, Re, In)$. If e is enabled, i.e $e \in \text{enabled}(G)$, we can execute e resulting in a DCR graph with new markings $G = (E, R, M')$, where $M' = (Ex', Re', In')$. The altered markings are obtained as follows:

$$\text{Ex'} \ = \text{Ex} \ \cup \ \text{e} \tag{4}$$

$$\text{Re'} \ = (\text{Re} \setminus e) \cup (e\bullet\!\rightarrow) \tag{5}$$

$$\text{In'} \ = (\text{In} \setminus (e\rightarrow\%)) \cup (e\rightarrow+) \tag{6}$$

Meaning that in (4), the altered executed marking is obtained by adding the executed event to the list of executed events. In (5) we obtain the altered pending marking by removing e from the list of pending events and add those events e target with a response relation. In (6) we obtain the altered include markings by first removing the events that e exclude on execution and the adding those events which e include on execution.

3.2.4 Definition 4: Transitions, runs and traces

We have a DCR graph $G = (E, R, M)$. If event e is enabled, $e \in \text{enabled}(G)$ and then executing e yields a new DCR graph H, we have that G has a **transition** on e to H. We can write this as $G \xrightarrow{e} H$.

A **run** of G is a, possibly infinite or finite, sequence of DCR graphs G_i and events e_i which results

in a sequence: $G = G_0 \xrightarrow{e_0} G_1 ...$
A **trace** is a sequence of labels of events e_i associated with a run of G. We can get the runs and traces of G by writing $runs(G)$ and $traces(G)$. A note to make is that a run or traces does not always result in an accepting graph (see definition 5).

3.2.5 Definition 5: Acceptance

We say a run $G = G_0 \xrightarrow{e_0} G_1 ...$ is accepting if and only if for all n with

$$e \in \text{ In}(G_n) \cap \text{ Re}(G_n) \tag{7}$$

there exists $m \geq n$ such that either $e_m = e$ or $e \notin \text{In}(G_m)$. Meaning that in (7) e is a included event which is also pending. The constraints inform us that in order for a graph to be accepting there must exist an m greater or equal to n so that either the transition event is the same or the event e is not included in later graphs in the run.

3.2.6 Definition 6: Language

The set of accepting traces for a DCR graph G is called the **language** of the graph. We write $lang(G)$ for the language.

3.2.7 DCR graph example

In this example we are looking at a simplified process of moving. Firstly, some possessions must be packed before any other activity in the process can occur. After the initial possessions have been packed, the moving truck can be loaded with the packed items. These 2 activities can occur until there are no more items to pack and the moving truck is loaded. When the moving truck is loaded and the activity of moving the possessions begins, one can no longer pack possessions, load the truck or move possessions and all that is left to do is unpack. We see the initial DCR graph in figure 1

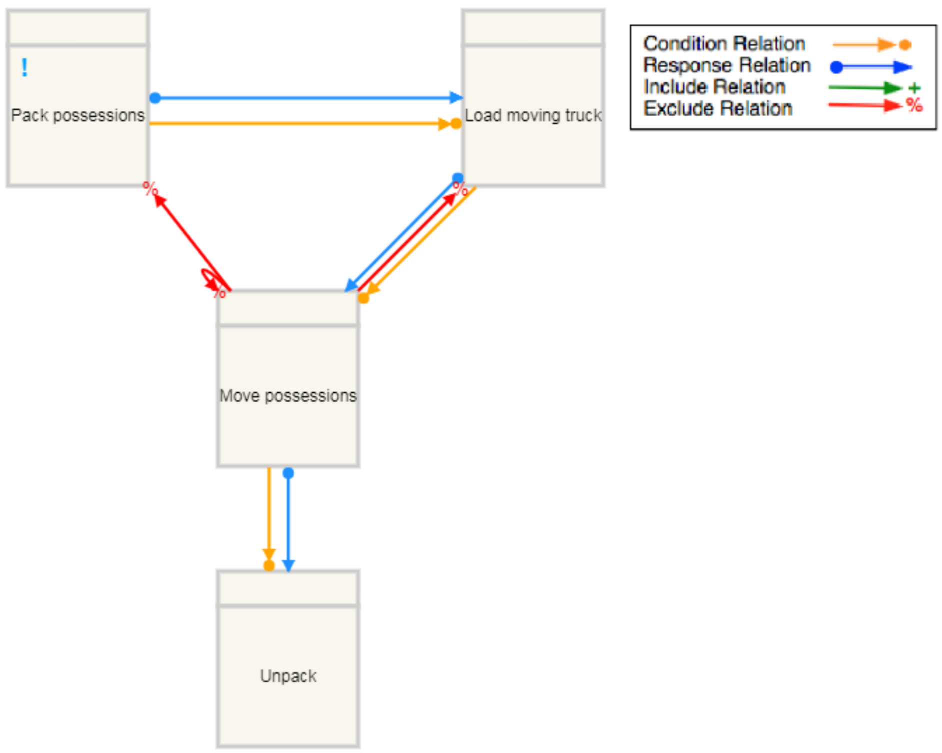
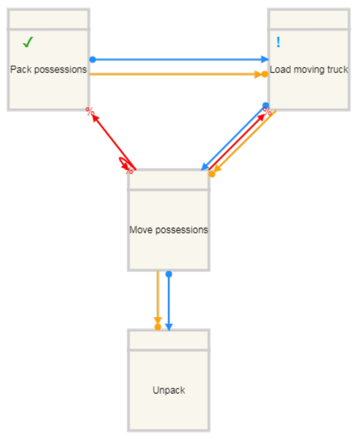
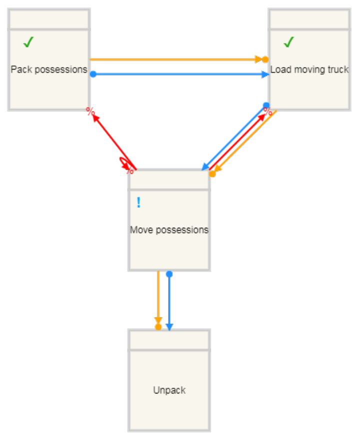


Figure 1: Initial DCR graph of the moving process

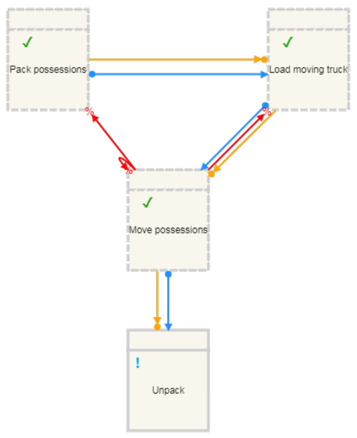
In Figure 2 we see the events in the graph being executed and in Figure 2d we see that the graph is accepting since there are no outstanding pending markings. In the initial figure 1, `Pack possessions` starts out as pending and has two relations to `Load moving truck`. These relations are a condition and a response.



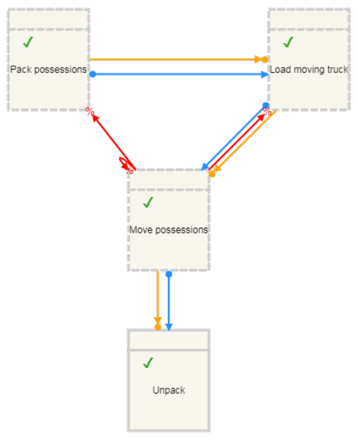
(a) "Pack possessions" event is executed



(b) "Load moving truck" event is executed



(c) "Move possessions" event is executed



(d) "Unpack" event is executed.

Figure 2: A small example of moving possessions.

When `Pack possessions` is executed, the pending status is removed and because of the response relation, `Load moving truck` is now pending and allowed to execute.

When `Load moving truck` is executed in 2b, the pending status is removed and `Move possessions` is now pending and allowed to be executed. At this state, `Pack possessions` and `Load moving truck` is allowed to be executed as many times as required, when they are done, the third event is executed.

In 2c when `Move possessions` is executed, the first two events, `Pack possessions` and `Load moving truck` are excluded along with `Move possessions` itself. `Move possessions` also makes the final event `Unpack` pending. When `Unpack` is executed, the pending status is removed and since there are no more pending markings on any of the events, the graph is accepting by Definition 5: Acceptance.

3.3 Implementing DCR graphs

We have implemented DCR graphs using object-oriented programming, using the language C#. Note that this section only details the parts necessary to implement DCR graphs, and excludes the parts of the implementation related to making DCR graphs distributed, which is instead detailed in section 4. Events are implemented as the *Event* object, and contain several different properties in order to represent their markings, a label, and a name which should be unique. These properties are:

- string Name
- string Label
- bool Include
- bool Pending
- bool Executed

Two new markings are also added as properties in order to represent the effect of blocking and unblocking execution of an event, that the Condition and Milestone relations have. These two properties are incremented by 1 for each event that are currently blocking them.

- int Condition
- int Milestone

Event objects have 5 fields, one for each relation type. These are made as HashSets and represent outgoing relations, and as such contain references to instantiations of other event objects. We used HashSets because relations are unique, meaning that there should not be more than one of the same relation between the same events. HashSets enforce this by preventing the same event from being added twice to the same HashSet, and also implements other useful methods to easily add and remove events - and check if a given event is in a HashSet: `Contains()`.
In order to determine whether an *Event* is enabled or not (see Definition 2: Enabled Events), *Event* objects implement the `Enabled()` method.

```
public bool Enabled()
{
    if (Included && Condition <= 0 && Milestone <= 0)
        return true;
    else
        return false;
}
```

Code Snippet 1: EventInternal.Enabled() method source code

A DCR graph is implemented as a *Graph* object. The events of a graph is contained in a `Dictionary` named `Events`, where each event is keyed by its name as a `string`. Events may be instantiated, and added to a graph by calling the graph's `CreateEvent()` method.

```
public void CreateEvent(string name, string label, bool included = true,
                        bool pending = false, bool executed=false)
{
    Events.Add(name, new EventInternal(name, label, included, pending, executed));
}
```

Code Snippet 2: Partial Graph.CreateEvent() method source code

This method simply allows defining the *name* of the Event used to index the aforementioned `Dictionary`, as well as optionally the initial markings of the event. Because dictionary keys must be unique, an exception will be thrown if the user attempts to add two events with the same name to the same graph, enforcing the rule that the `Name` property should be unique.
After adding at least two events to a graph, relations may be added by using the *Graph* class method `AddRelationInternal()`.

```
public void AddRelationInternal(string eNameFrom, string eNameTo, Relation relation)
{
    EventInternal eFrom = Events[eNameFrom];
    EventInternal eTo = Events[eNameTo];
    switch (relation)
    {
        case Relation.Response:
            eFrom.Responses.Add(eTo);
            break;

        case Relation.Condition:
            eFrom.Conditions.Add(eTo);
            if (eFrom.Included && !eFrom.Executed)
                eTo.Condition++;
            break;

        case Relation.Milestone:
            eFrom.Milestones.Add(eTo);
            if (eFrom.Included && eFrom.Pending)
                eTo.Milestone++;
            break;
    }
}
```

Code Snippet 3: Partial Graph.AddRelationInternal() method, excluding Relation.Include and Relation.Exclude case, as they are nearly identical to the Response case.

As seen in the code, we first obtain a reference to the events that the desired relation is from and towards: `eFrom` & `eTo`. This is done by supplying their names as strings as parameters to the method, and using these to index the `Events` dictionary. Additionally, a new enumerated type `Relation` is used and supplied as argument to the method, which represents the type of relation to be added. This type is then matched using a switch statement, and added to the `HashSet` in

eFrom. If the relation is a Condition or Milestone, we also sometimes increment the `Condition` or `Milestone` property of `eTo`, depending on the current marking of `eFrom`.
With the ability to create a graph, and add events and relation to this graph, we now want to be able to execute events. For this, the `Graph` class has the `Execute()` and UpdateMarkingsInternal() method:

```
public bool Execute(string eventName)
{
    EventInternal ev = Events[eventName];

    if (!ev.Enabled())
        return false;

    UpdateMarkingsInternal(ev);
    ev.Executed = true;
    ev.Pending = false;
    return true;
}
```

Code Snippet 4: Partial and slightly modified Graph.Execute() method, excluding P2P, parallelism, and logging capabilities

Since we do not update the properties of the executed event prior to calling `UpdateMarkingsInternal()`, we are able to discern the exact way in which the other events, that the executed event has relations to, should be updated. This is especially crucial as changes to the markings of these related events may result in even more changes, as detailed below.

```
private void UpdateMarkingsInternal(EventInternal ev)
{
    foreach (EventInternal e2 in ev.Responses)
    {
        if (e2.Included && !e2.Pending)
            foreach (EventInternal e3 in e2.Milestones)
                e3.Milestone++;

        e2.Pending = true;
    }
    foreach (EventInternal e2 in ev.Excludes)
    {
        if (e2.Included && ev != e2)
        {
            if (!e2.Executed)
                foreach (EventInternal e3 in e2.Conditions)
                    e3.Condition--;

            if (e2.Pending)
                foreach (EventInternal e3 in e2.Milestones)
                    e3.Milestone--;
        }
        e2.Included = false;
    }
    if (!ev.Executed && ev.Included)
        foreach (var e2 in ev.Conditions)
            e2.Condition--;
    if (ev.Pending && ev.Included)
        foreach (var e2 in ev.Milestones)
            e2.Milestone--;
}
```

Code Snippet 5: Partial UpdateMarkingsInternal(), excluding the code for the Included relation

Since executing an event with a response relation will make the impacted event pending, we check whether or not it is already pending. We then check if it is also included and has any milestone relations, and if it does, we increment the `Milestone` property of the impacted event e_3. If it was already pending, a response relation will have no effect, and we do nothing.
Next, if we exclude an event e_2, which is a condition for another event e_3, we need to update the `Condition` property such that e_3 is no longer blocked for execution, by decrementing it by 1. In case e_2 has already been executed prior to the current execution, the condition relation loses its effects and we should not do anything. Similarly to this, if e_2 is pending, we check whether it has any milestone relations, and if they do, unblock e_3 by decrementing its `Milestone` property. Finally, we set e_2's `Included` property to false. It is worth noting here, that e_1 might have an Exclude relation to itself, meaning $e_1 = e_2$ and thus e_1.Included = false. This means that if e_1 has a Condition or Milestone relation to some e_2, and has not been executed before, we ensure in the next two If-statements that we have not already decremented e_2's `Condition` or `Milestone` property in the prior `foreach` loops, where e_3 in that loop would be our current e_2 in the if-statements.
Finally, we determine whether a graph is in an accepting state, as per the definition given in Definition 5: Acceptance, with the *Graph* class method. `GetAcceptingInternal()`:

```
public bool GetAcceptingInternal()
{
    foreach (EventInternal e in Events.Values)
    {
        if (e.Pending && e.Included)
            return false;
    }
    return true;
}
```

Code Snippet 6: GetAcceptingInternal() method

If the graph is not accepting, we return a bool indicating that the result is false, i.e not accepting. The method checks if any event in the graph is pending and included, according to the formal definition, Definition 5: Acceptance.

4 DCR graphs in a distributed setting

In this section, we will start by explaining the network architecture we chose in order to distribute these DCR graphs across different systems, as well as some of the challenges we faced and design decisions we had to make. We will then detail the communication protocol we used in the aforementioned network architecture, show some examples of this communication, as well as relate to the communication protocol used in [2]. Finally, we will present our implementation of a distributed peer-to-peer DCR graph engine.

4.1 Network Architecture

In our peer-to-peer network we decided to split the network into two parts - Main Nodes and Nodes. Communication between Main Nodes is the traditional pure peer-to-peer system, while Node to Main Node communication works as a client-server system. The reason for this split will be outlined in subsubsection 4.1.1, where we detail the different challenges that we faced while designing and implementing the system.

Each Main Node contains a local DCR graph and is responsible for handling the associated logic, i.e executing an event or keeping track of whether or not an event is enabled. The relations between events are also stored in the Main Node and each Node belongs to a Main Node, where they act as events in the local graph. Main Nodes can send and receive to and from other, and each Main Node has port forwarding enabled in order to receive incoming messages.

Nodes can send to Main Nodes, and receive a reply to their message, but Main Nodes cannot initiate a connection to a Node, due to Nodes not having port forwarding enabled. In this way, Main Nodes act as a proxy for the Nodes, allowing Nodes to manipulate their associated Main Nodes local DCR graph, and extract data from it and the entire, global DCR graph. If a Node requests information from a Main Node that is not available locally, the Main Node will propagate the request on the network, gathering all the information before finally sending it back as a reply to the initial request. Nodes cannot communicate with each other directly, but a node may manipulate the graph, and these changes will be visible to the other nodes in their interaction with the Main Nodes.

Our design is illustrated in Figure 3, where each Node represents a user, and the Main Nodes represent servers, which do not take any actions until they receive a request from a Node or another Main Node - which would also have originated from a Node's request.

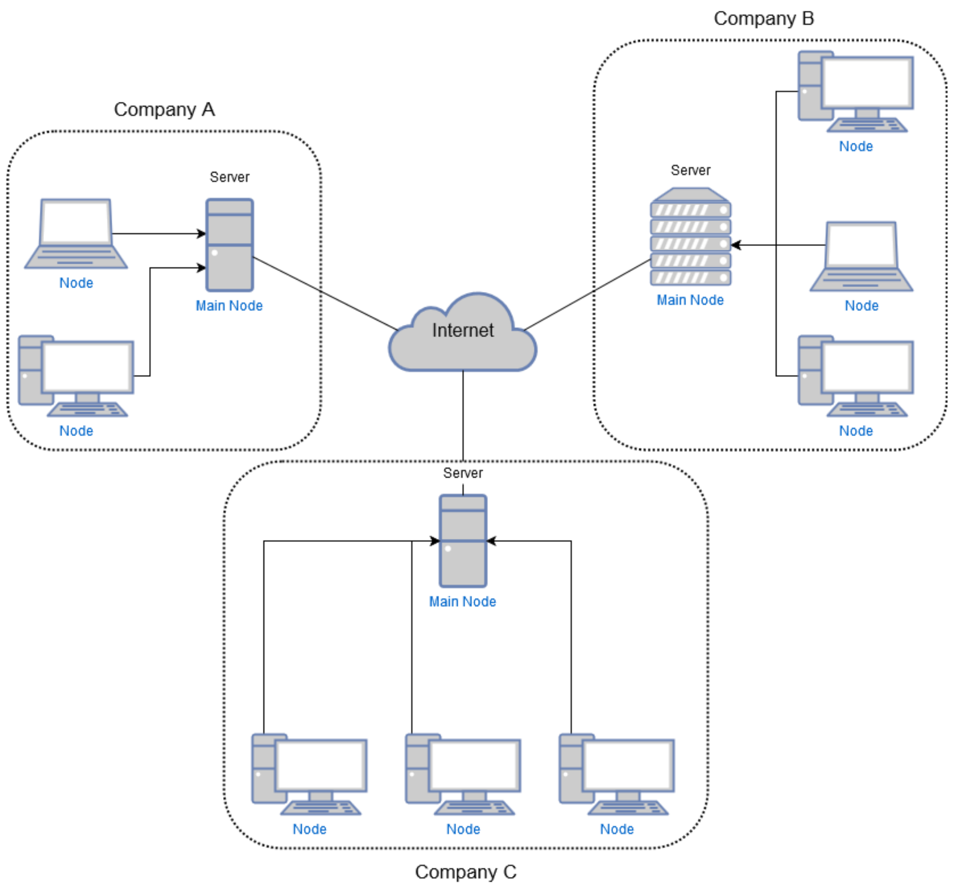


Figure 3: Network architecture example

An example is shown in Figure 4. If a Node wishes to execute its event e_1 located on a Main Node m_1 and the event affects an event e_2 located on another Main Node m_2, the Node sends the execution request to m_1. m_1 then checks if the event is enabled, and if it is, contacts m_2 to check if e_2 is currently in a blocked state. The blocked state is synonymous with a lock, ensuring that the event can not be changed by other events than the execution event, see subsubsection 4.2.1. If it is, m_2 replies that it is unavailable, otherwise, it blocks it, disallowing manipulation of its state by anything other than the current execution of e_1. Upon receiving the message that the block was successful, m_1 tells m_2 to change e_2's marking to reflect the consequences of executing e_1, and finally replies that the execution of e_1 was successful to the node that requested it.

Changing the marking of e_2 in the case above, may also change the marking of additional events. For example, e_1 may have a Response relation to e_2, resulting in e_2 becoming pending in the example. If e_2 is both Included and has a Milestone relation to some other event e_3, which is located on a third Main Node m_3, then m_2 should, as described earlier, send a request to m_3 in order to try to block e_3. If this fails, m_2 should report that it is unavailable to m_1, which will then reply to its local Node that the execution could not happen because the events are currently unavailable.

If we did not do this, we could end up changing the marking of e_2 to be pending, which should block the execution of e_3 due to e_2's Milestone relation, but because e_3 is not blocked, it might have been executed before the markings were updated. By applying a block, markings are propagated securely in the network. Main Nodes m_1 has a reference to e_2, and m_2 has a reference to e_3, which their local, also referred to as "internal" events, have external relations towards. That is to say, that m_1 does not know or update the current state of event e_2 locally, but merely uses its reference to it in order to propagate markings. Note also, that while m_1 and m_3 can communicate back and forth with m_2, they do not communicate directly with each other, and do not need to know that each other is part of the network.

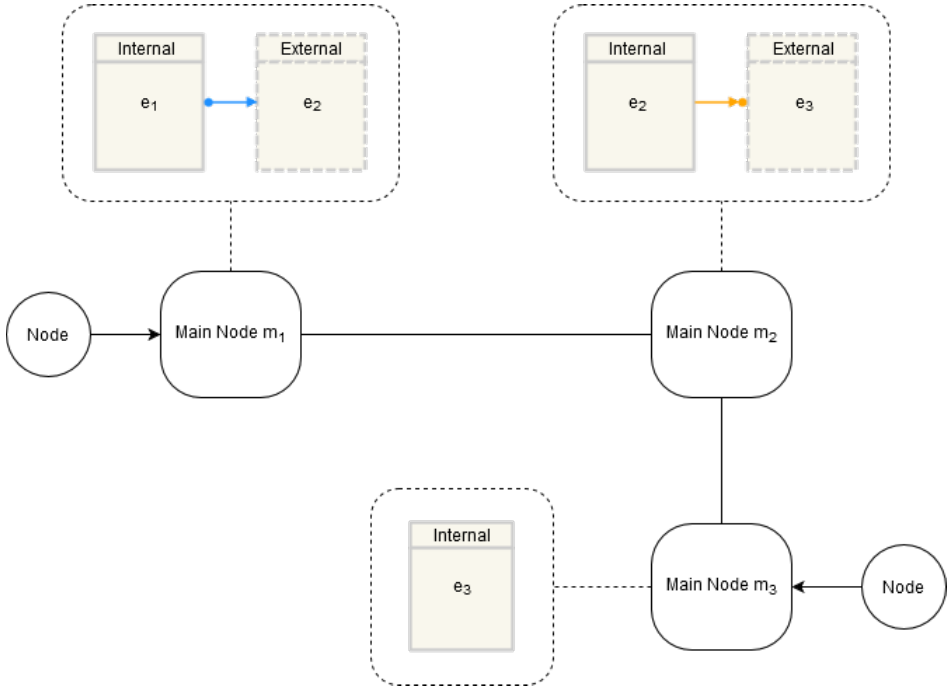


Figure 4: Distributed DCR graph example

4.1.1 Network architecture challenges

Our initial approach to creating a peer-to-peer system was close to our current system, except every node had the Main Node architecture, and no intermediaries. This quickly proved to be an unrealistic architecture, as regular peers, such as some workers laptop or personal computer, would not have port forwarding enabled by default, as it would otherwise pose a security risk. We outlined several solutions to this problem:

- Universal Plug and Play (UPnP)
  UPnP is a set of networking protocols that allow devices to automatically forward ports whenever it is needed. This can be used to seamlessly establish a peer-to-peer connection, as long as one of the peers have UPnP enabled. This does, however, pose some security risks. For example, a typical peer in our distributed system would be a regular employee's laptop, which does not have the security or reliability that e.g. a company server does. Ensuring high reliability and availability for e.g. a company hosted server is much more realisable than doing the

same for your employees' devices. It would also be much more difficult to implement the same standard of security in terms of both network- and application-layer firewalls as a company server. Additionally, while UPnP is convenient, the very nature of allowing peers to open ports, as well as other UPnP specifically, has a history of being exploited by hackers [7].

- UDP hole punching
  UDP hole punching is a NAT(Network Address Translation) traversal technique that is used to connect two peers to each other via a rendezvous server as in [8]. The technique is most commonly used when two peers need to connect to each other but there is a NAT between them that is stopping the connection from being established. The two peers can trick the NAT by first establishing a connection to the server and then exchanging the server's endpoint with the other peer's endpoint. When the connection is established to the server, an entry is added to the NAT table and when the endpoint are exchanged the entry in the table is changed to the peers endpoint. The NAT will allow this, since no new connections are being made but an existing entry in the table is being altered.

  For this to properly function in the peer-to-peer DCR graph engine, there would have to be a dedicated server for these types of connections. This server would help in discovery of new peers in the network and the aforementioned establishing of connections. The introduction of the server in the network would change it from being a pure peer-to-peer to a hybrid peer-to-peer network, since the peers no longer connect directly to each other. If the server were to crash or fail the peers would not be able to connect to each other, this highlights the server as a single point of failure in the network as the network essentially stops when the server does. If the system is used in a company setting, the ownership and management of the server would become a point of contention, since the server would have to be in one of the companies. The company who hosts the server would have access to it and they could perform malicious actions with the server, which would be difficult for the other companies to discover. There also exists a TCP hole punching technique, which is slightly more complicated than the UDP version and produces a TCP connection.

- Port-forwarded Main Nodes
  Another solution would be to use peers with port forwarding enabled as proxies for the non-port forwarded peers (Nodes) to communicate with. As previously mentioned, attaining a high level of reliability, availability, and security of e.g. a single server per company is easier than doing it for the device of each employee in a company. Here, we envisioned the servers as Main Nodes in the network, containing the local states of the DCR graph and being responsible for both receiving requests from other Main Nodes outside the network, and sending requests on behalf of employees. Regular nodes, as in employee devices, are then simply used to request their local Main Node.

We ended up choosing the Port-forwarded Main Nodes design because it compliments the fact that

companies usually have one or more servers with port forwarding already enabled, that employees use to communicate with the public internet. This also allows companies to implement security measures and use their already existing measures of their server, exposing only some functionality of the DCR graph to the outside of the network using a DMZ (demilitarized zone), and allowing their employees access to executing events within the LAN (local area network).
Another challenge was how to propagate markings throughout the network upon executing an event. Initially, we figured that every Main Node would know each other, and thus be able to update markings. This posed a problem, however, as problems arise in cases such as in the example from subsection 4.1. The problem lies in, that a Main Node is not aware of the marking of any external events, and thus does not know whether it should send additional blocking requests, such as the case in the previously mentioned example, where e_2 is both Included and becomes Pending, and has a Condition relation to e_3. Thus, the responsibility of blocking e_3 should rest on m_2 instead of m_1, which cannot infer the marking of e_2. Solving this challenge also removed the necessity for every Main Node being connected to each other, removing a lot of communication overhead.

We faced an additional challenge with the *TcpClient*'s implementation of the TCP protocol in when the client Main Node wanted to disconnect, it would hang on the call to `Disconnect()`. We investigated this issue by using Wireshark[1] to inspect the packages sent and received by the client and the receiver and discovered that the client did not receive an ACK for its FIN package. When disconnecting in TCP, the initiator of the disconnect sends an FIN package and the receiver replies with an ACK before sending a FIN of its own. The initiator then sends an ACK in return and the connection is closed. In our implementation, the initiator sometimes did not receive an ACK for its FIN and thus the connection was not closed. This resulted in temporary standstill in the execution where the `Disconnect()` method waited until it timed out by sending a RST package, representing a reset before closing the connection. The result can be seen in Figure 5, where there is no response to the [FIN, ACK], and then an [RST, ACK] is sent.

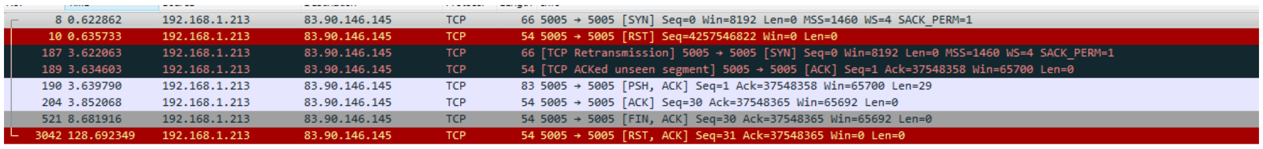

```
   8 0.622862     192.168.1.213   83.90.146.145   TCP   66 5005 → 5005 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
  10 0.635733     192.168.1.213   83.90.146.145   TCP   54 5005 → 5005 [RST] Seq=4257546822 Win=0 Len=0
 187 3.622063     192.168.1.213   83.90.146.145   TCP   66 [TCP Retransmission] 5005 → 5005 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
 189 3.634603     192.168.1.213   83.90.146.145   TCP   54 [TCP ACKed unseen segment] 5005 → 5005 [ACK] Seq=1 Ack=37548358 Win=65700 Len=0
 190 3.639790     192.168.1.213   83.90.146.145   TCP   83 5005 → 5005 [PSH, ACK] Seq=1 Ack=37548358 Win=65700 Len=29
 204 3.852068     192.168.1.213   83.90.146.145   TCP   54 5005 → 5005 [ACK] Seq=30 Ack=37548365 Win=65692 Len=0
 521 8.681916     192.168.1.213   83.90.146.145   TCP   54 5005 → 5005 [FIN, ACK] Seq=30 Ack=37548365 Win=65692 Len=0
3042 128.692349   192.168.1.213   83.90.146.145   TCP   54 5005 → 5005 [RST, ACK] Seq=31 Ack=37548365 Win=0 Len=0
```

Figure 5: Wireshark result

To solve this issue, we ended up overhauling our communication protocol. Instead of reusing the same socket over and over for each Main Node, and therefore having to wait till it is disconnected properly before being able to reuse it again, we use the first available port to open sockets, and use UTF-8's "END OF TRANSMISSION" character in order to determine when a sequence of messages is finished. An example of a sequence of messages could be a BLOCK, which is typically followed by an EXECUTE, and finally an UNBLOCK. By appending the "END OF TRANSMISSION"

[1]Wireshark.org

character to the last message in such sequences, the sender and receiver are able to agree when to stop reusing a socket, and instantly discard it without having to wait for a disconnection to finish. Any new sockets can then be created by picking the first available port.

4.2 Communication protocol

The implementation of peer-to-peer functionality is split into two parts in the Main Node - a client and a listener. The Client is responsible for all outgoing communication with other Main Nodes and the Listener handles incoming messages from Main Nodes and Nodes. Both parts are implemented with the TCP protocol, which supplies a range of features such as reliable transmission, error detection and flow control. The client is implemented with the *TcpClient* class, which provides simple methods for sending and receiving to and from a *TcpListener* class. The client binds itself to a socket on the computer and uses it to open a *NetworkStream* to the Listener. The *NetworkStream* transmits messages between two endpoints by sending a byte of the message at a time. This means that the entire message needs to be converted into an array of bytes before the *NetworkStream* can transmit it. When the message has been received, it has to be converted to the original type in order for it to be interpreted by the receiver.

In order to convert to and from strings and bytes, we use the C#'s `Text.Encoding` library. On the client side, the UTF-8 encoded string message is converted to bytes and stored in the aforementioned array before it is sent. On the listener side, these bytes are received and converted back to UTF-8.

The Listener is implemented with the *TcpListener* class and listens for incoming TCP connection requests on a specified, forwarded port. If the targeted port in the incoming request is not a forwarded port, the request is automatically denied. When a new valid TCP request arrives, the Listener binds the request to a client and prepares to receive the incoming message. The Listener stores the incoming message of bytes in a byte array, which is converted to a string and passed on to the *Parser*.
The Listener runs on a separate thread from the main thread. This ensures that the Listener can always handle incoming requests without blocking the main thread.

4.2.1 Message types

Messages between Main Nodes and from Nodes are converted from bytes to strings, and these strings are then interpreted by a parser. An "N:" or "M:" is prepended to every message, indicating for the receiver whether the sender is a Node or Main Node. The different types of messages that Main Nodes may send and receive are:

- BLOCK
  - BLOCK messages are sent and propagated in order to block external events from being

executed or having their markings changed. A BLOCK message is initially sent by a Main Node that has been prompted to execute an event, which will change the markings of any number of external events.

  - The BLOCK message includes information about which local event of the receiving Main Node that needs to be blocked, as well as which marking(s) needs to be changed if execution is successful.
  - To allow blocked events from being executed by the execution that is blocking them, and indicate which events are blocked by what ongoing execution, a globally unique identifier (GUID) is also supplied with this message.
  - If blocking succeeds, the Main Node will reply with "SUCCESS". Otherwise, it will reply with "UNAVAILABLE".
  - The BLOCK message is named Block since it blocks an event from being executed by other events. It is essentially a lock on an event, ensuring that it is available when the executing event needs it to be.

- EXECUTE
  - An EXECUTE message is sent after all events have been successfully blocked, meaning all Main Nodes have replied "SUCCESS" to a BLOCK message.
  - Since all information regarding which markings to change has been supplied with the BLOCK message, an EXECUTE message only contains an identical GUID to the one provided in the BLOCK message.
  - Main Nodes reply "SUCCESS" if markings were successfully changed, or "UNAVAILABLE" if the provided GUID does not match any currently active GUID.

- REVERT
  - If one or more recipients of an EXECUTE message replies "UNAVAILABLE" or fails to respond, A REVERT message is sent to the recipients who replied SUCCESS in order to revert the changes in markings that EXECUTE resulted in. It also unblocks the Main Node.
  - Revert messages also include an identical GUID to the one provided in BLOCK, in order to indicate what execution is supposed to be reverted.
  - Similarly to EXECUTE, Main Nodes will reply either "SUCCESS" or "UNAVAILABLE".
  - A REVERT message is only rarely required because of our blocking mechanism, but may sometimes be neccessary e.g. due to a Main Node crashing or having internet problems. Here, REVERT ensures that all the Main Nodes's markings which were changed by the prior execution, are reverted to their previous state, thus preserving the synchronisation of all Main Node's events - including the one(s) that were unavailable.

- UNBLOCK
  - An UNBLOCK message is sent after receiving a "SUCCESS" reply to an EXECUTE message from all messaged Main Nodes.
  - It may also be sent if a Main Node is unavailable upon sending a "BLOCK", where it will then be sent to everyone that blocked successfully, in order to unblock them.
  - Like before, a GUID is included in the message.
  - Main Nodes will reply "SUCCESS" or "UNAVAILABLE" just like described in EXECUTE and REVERT.
- ACCEPTING
  - This message is used to determine whether a graph is in an accepting state, by asking each Main Node whether their local state is accepting.
  - An ACCEPTING message is accompanied by the names of the Main Nodes that have already been sent an ACCEPTING for the current query.
  - If the local state is accepting, the message is propagated to all known Main Nodes that have not already received one (although it is not possible to account for all cases, as described in subsection 4.3).
  - A Guid unique to the current ACCEPTING query is also provided and used to prevent propagating the ACCEPTING message more than once. If this is the case, the Main Node will reply "ACCEPTING IGNORE" to indicate the sender that it should ignore the reply.
  - Main Nodes will reply "FALSE" if its own or any of the queried Main Nodes reply "FALSE" to the propagated message. Otherwise, it will reply "TRUE", indicating that all queried Main Nodes are in an accepting state.
- LOG
  - A LOG message is sent and propagated by Main Nodes when a Node requests to get a Log containing all executions that have happened in the graph, along with their timestamps.
  - Logs are propagated in a similar way to ACCEPTING messages, and duplicates are ignored similarly with a GUID and a "LOG IGNORE" reply.
  - If the requested Main Node's gathered log - including the propagated requests - is empty, it will reply with a "LOG EMPTY" message.
  - Each Main Node will propagate the LOG request before finally replying with a string containing a list of all of the gathered executed events and timestamps. Finally, when the entire log has been gathered, the Main Node which was the recipient of the first LOG message replies with a global, sorted log to the Node.

As Nodes do not have port forwarding, they never receive any messages from each other or Main Nodes unless it is a reply to a request that they have made. The requests that they may make depend on their privileges given to them by their assigned Main Node. For example, a Main Node may allow node n_1 to execute an event e_1, but not e_2, which is only executable by n_2. To find out which events are executable by a given node, it may send a message asking for its permissions.
All nodes may, however, request a global log of executed events, whether the global graph is in an accepting state, and get the marking of any and all local event(s).

- PERMISSIONS
  - Requests a string with the names of what events the Node has permissions to execute. This is not whether they are executable or not, but simply which events the Node may attempt executing.
- EXECUTE
  - Executes a specified event. Has to occur after at least one "PERMISSIONS" message, as the Node will only query for executions of events that it locally knows it may execute. This was done in order to lower network usage.
- ACCEPTING
  - Requests whether the global graph is in an accepting state.
- LOG
  - Requests a global, sorted log of executed events along with their timestamps.
- MARKING
  - Requests the marking of a specified, local event.
- ALLMARKINGS
  - Queries the Main Node for the markings of all the events belonging to the Node.

4.2.2 Communication examples

In Figure 6 is an example of how messages are propagated between Main Nodes. This uses the "ACCEPTING" message as an example, but the same overall structure for propagating messages is used for the other messages.

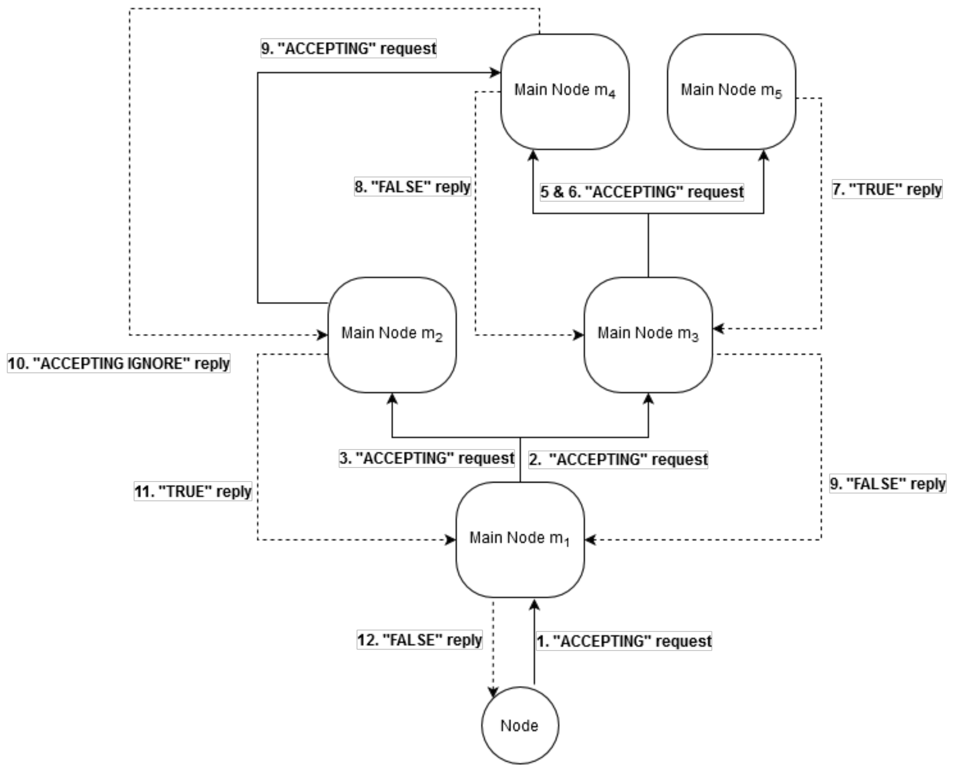


Figure 6: "ACCEPTING" request in a network with 5 Main Nodes

In this example, we see that all local graphs except the one belonging to m_4 is in an accepting state. The messages are ordered with a number indicating the time that they are sent. As described in subsubsection 4.2.1, an "ACCEPTING IGNORE" message is sent as a reply to an "ACCEPTING" request containing an already known - and therefore received - GUID. This can be seen in **9** and **10**, where m_2 sends an "ACCEPTING" request with a GUID that has already been sent by m_3. As a result, m_2 replies with "TRUE" to m_1 in message **11** even though m_4 is not in an accepting state. This is no problem, however, as a "FALSE" from m_4 is propagated back to the Node by m_3 in **8**, **9** and **12**.

In Figure 7 and Figure 8, we will show how events are executed and their markings are propagated. The first example shows an event with markings that need to be propagated twice, being executed without any mishaps - meaning that none of the Main Nodes are unavailable for any reason.

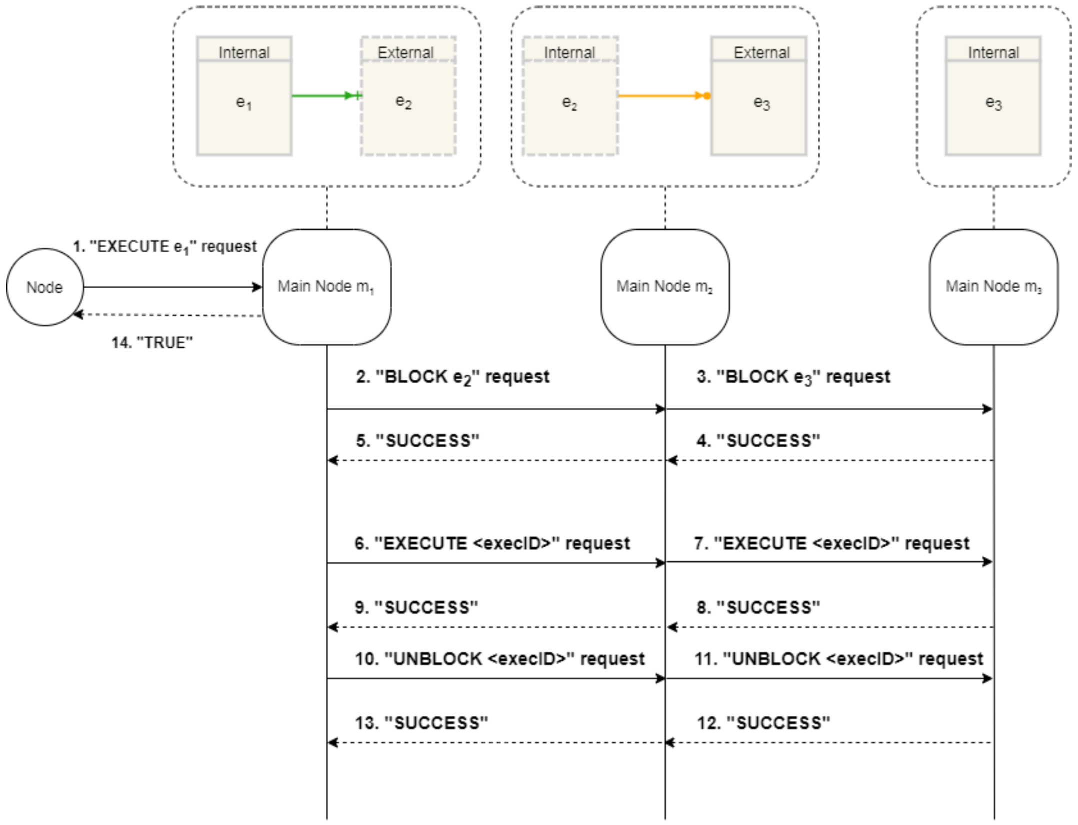


Figure 7: A successful "EXECUTE" request in a network with 3 Main Nodes.

Here we see that m_1 has an e_1 with an external Include relation to e_2, which is located in the m_2 Main Node. Additionally, e_2 has an external Condition relation to e_3 located in m_3. In the example, e_2's Included marking is false. This means that it is not currently blocking e_3 with its Condition relation. When e_1 sends a "BLOCK" request, it informs e_2 that it has an Include relation to e_2. From this, m_2 can infer that changing in e_2's Included marking to true will result in a change of e_3's marking as well, and it therefore propagates the "BLOCK" message to m_3, where it informs m_3 that the condition marking of e_3 is about to change. Thus, m_3 blocks e_3 and replies with a "SUCCESS" message to m_2, which in turn, replies back to m_1 with a "SUCCESS" message. Upon receiving this message, m_1 knows that it is safe to execute, and does so by messaging "EXECUTE" to m_2, which propagates this to m_3. After m_2 gets a reply from m_3 indicating the markings were changed successfully, m_2 changes its own marking such that e_2 is now included.
When m_1 receives a "SUCCESS" reply from all the other nodes, in this case only m_2, it also executes and changes the marking of its own local events. In this case, the only marking that is changed is the Executed marking of e_1. Note that all events are still blocked, as we previous did not know that the execute would be successful at all Main Nodes. Since we now are, we send an "UNBLOCK" message to e_2, which is propagated to e_3. This unblocks the events, finalising their marking and allowing their marking to be changed once more by other unrelated executions. The execution of e_1 is now complete, and m_1 replies with "TRUE" to the node that requested the execution.

In the second example, we see what happens in case a Main Node m_4 is unavailable after an "EXECUTE" message has been sent, resulting in a "REVERT" message being sent.

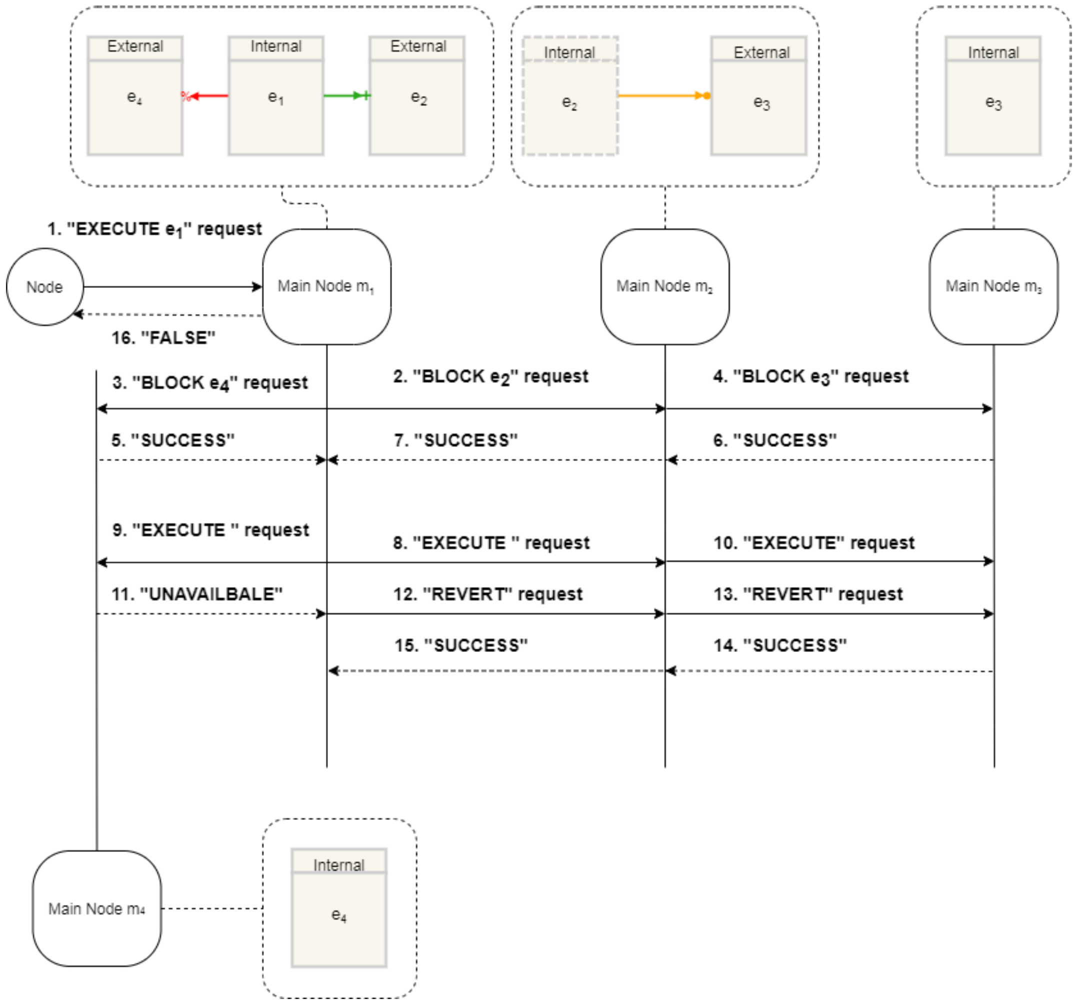


Figure 8: A unsuccessful "EXECUTE" request in a network with 4 Main Nodes

This example is similar to the previous, except the network has an additional Main Node m_4, with an external Exclude relation from e_1 to e_4. The steps of e_1's execution also go as explained in the previous example until m_4 either replies "UNAVAILABLE", or does not respond to the "EXECUTE" request at step **9**. Since the execution could not be completed, m_1 sends a "REVERT" message to m_2, who propagates it to m_3. In this case, m_3 decrements the Condition marking of e_3 by one, replies "SUCCESS" to m_2, who then proceeds to set e_2's Included marking back to False. Finally, it returns "FALSE" to the node that requested the execution.

4.2.3 Marking based propagation

In order to communicate changes in the DCR graph, we decided to propagate the changes in markings by messaging these changes to the owner of the event(s). As we will describe in more detail in

subsection 4.3, this is done by both straightforwardly communicating which markings to change by mapping outgoing relations and events locally, as well as sometimes making a temporary local copy of an external event. When that event has a relation which is going to change a local, internal event, this copy is then discarded after execution.
This is a different approach from [2][2], which presents another method to project a DCR graph, by having a permanent copy of every external event that affects a local event. When an event executes, it will communicate to the other graphs containing copies of the event to execute the copy as well. Thus, communicating execution. Our method of communicating markings uses the technique of creating a temporary external event in a local graph. The reason for this is that we use the temporary external event as a container for the markings that should be changed between the different messages being sent back and forth, and after the execution is complete, the temporary event has no use. The temporary external event is never actually executed in the local graph, but markings are changed based on what is saved inside the event. In order to ensure the global state of the graph, we have also made a way to revert marking changes if the need should arise.

4.3 Implementing distributed DCR graphs

See section 7 for setup instructions for the graph, node and network.

4.3.1 Config

Config implements the *IConfig* interface in order to reduce coupling between the different classes, which is elaborated in section 5.
Config is the first class to be instantiated in the system, since all the other modules like *Graph* and *Log* use it. When the class is instantiated, it reads from the configuration file whose name and path is given as parameter in the constructor. The *config* class contains general networking information for the local graph, its nodes and its known Main Nodes. For example, the *Config* specifies which port the `Listener` should listen to and how many connection attempts is allowed for re-connecting to an unresponsive Main Node. We can see the fields of *Config* in **Code Snippet** 7.

```
private string name;
private int listenPort;
private int clientTimeoutMs = 10000;
private int maxConnectionAttempts = 5;
private CultureInfo culture = CultureInfo.CreateSpecificCulture("en-GB");

private Dictionary<string, IClient> mainNodes = new Dictionary<string, IClient>();
private Dictionary<string, IPAddress> nodes = new Dictionary<string, IPAddress>();
private Dictionary<IPAddress, HashSet<string>> nodeEventsMapping =
                new Dictionary<IPAddress,HashSet<string>>();
```

Code Snippet 7: Config Properties

[2]"Safe Distribution of Declarative Processes" p. 7-8

In the `mainNodes` dictionary, we keep track of known Main Nodes by having the key to the dictionary be their identifier as a string. This allows us to index any Main Node in constant time, and get their *Client* instance whenever we want to send messages to them. The `nodes` Dictionary likewise keeps track of known Nodes, but we only need to know their IP address, and do not require the functionality of the *Client* class. This is because Nodes do not have port forwarding, meaning we will never need to initialize communication with a Node with e.g. `Client.SendAsync()`, and only reply to requests from it. These properties are filed out by the config file, with the file extension of `.ini`. A typical config file will look as following:

```
[Properties]
Name=ThisMainNodeName
ListenPort=5005
ClientTimeoutMs=10000
MaxConnectionAttempts=5
Culture=en-GB
[MainNodes]
MainNode1=5.186.127.236:5006
MainNode2=192.168.0.27:5007
[Nodes]
node1=127.0.0.1
node2=127.0.0.2
node3=127.0.0.3
[NodeEventsMapping]
node1=e1
node2=e2
node3=e1 e3
```

Where the lines containing hard brackets as "[Properties]", it indicates that the following is of a specific category. At the moment there are four categories: [Properties], [MainNodes], [Nodes] and [NodeEventMappings]. [Properties] relate to the general network setup. [MainNodes] define the name of the Main Nodes for this current local Main Node, along with the IP and port that should be used to communicate with them. [Nodes] defines which Nodes this local Main Node has and [NodeEventMapping] is which Nodes has access to which events. A more detailed explanation of how to configure this file is provided in section 7.

4.3.1.1 Constructor

The constructor of *Config* takes a string, representing the path to the .ini file, as input. It then attempts to read the file at the specified location and if no file is found it will throw an exception. This exception is purposefully left unhandled, indicating that a serious error has been made, crashing the system. The reasoning for this, is that the .ini file contains most of the information for the graph and if no .ini file is found, the graph can not be made. Next, it uses a `StreamReader` to read the entire .ini file, splitting the file by newlines and removing empty entries and saving these in a string array, called `lines`. It then loops through `lines`, using a while-loop and an integer `i`, where `i` represents the current line of the file. The while-loop stops when `i` reaches the number of entries in `lines`, indicating that there are no more lines to read. The body of the while-loop is a switch statement which matches on "[Properties]", "[MainNodes]", "[Nodes]" or "[NodeEventMapping]". If none of these are found it will increment `i` by one and try the next line. If one of the aforementioned cases occur it will call a corresponding method to the case. I.e. if "[MainNodes]" occurs, it will call `SetMainNodes`, which will return a new value for `i`, indicating a jump in lines when reading. When `i` is equal to the number of lines in the string array the while loop will terminate and the constructor is done. The code is shown in **Code Snippet** 8.

```
public Config(string path)
{
    StreamReader reader = new StreamReader(path);
    string file = reader.ReadToEnd();
    reader.Close();

    string[] lines = file.Split(new string[] { "\n", "\r\n" },
        StringSplitOptions.RemoveEmptyEntries);
    int i = 0;
    while (i != lines.Length)
    {
        switch (lines[i])
        {
            case "[Properties]":
                i = SetProperties(lines, i + 1);
                break;
            case "[MainNodes]":
                i = SetMainNodes(lines, i + 1);
                break;
            case "[Nodes]":
                i = SetNodes(lines, i + 1);
                break;
            case "[NodeEventsMapping]":
                i = SetNodeEventsMapping(lines, i + 1);
                break;
            default:
                i++;
                break;
        }
    }
}
```

Code Snippet 8: Config constructor

4.3.1.2 SetMainNodes

`SetMainNodes()` is very similar to `SetNodes()` and `SetNodeEventsMapping()` used in the config constructor, and the two lastly mentioned methods are therefore omitted.
`SetMainNodes()` take a string array, `lines`, containing the lines of the .ini file as input, along with an integer `i` indicating which line to start at. The method then enters a while-loop where the conditions are that `i` is not equal to the length of lines and that the first character of the line is not an opening hard-bracket([). If these conditions hold it will split the lines into three, the name, IP and port of the Main Node. When these are found it will create a `Client` class with the IP, Port and itself and save this client in the configs' `mainNodes` dictionary with the name as key and client as value. It will then increment `i` and check if there are more entries for Main Nodes, if not it will return `i`.
The code is shown in **Code Snippet** 9.

```
private int SetMainNodes(string[] lines, int i)
{
    while (i != lines.Length && lines[i][0] != '[')
    {
        int ipStart = lines[i].Substring(0, lines[i].IndexOf('=')).Length;
        int portStart = lines[i].Substring(0, lines[i].IndexOf(':')).Length;

        string mName = lines[i].Substring(0, ipStart);
        string mIP = lines[i].Substring(ipStart + 1, lines[i].Length
            - (lines[i].Length - portStart) - ipStart - 1);
        string mPort = lines[i].Substring(portStart + 1);

        Client client = new Client(mIP, Convert.ToInt32(mPort), this);
        mainNodes.Add(mName, client);
        i++;
    }
    return i;
}
```

Code Snippet 9: SetMainNodes

4.3.2 Graph

The properties of the *Graph* class pertaining to non-distributed functionality have already been described, and will thus be omitted from this section. *Graph* does, however, make use of many different lists and Dictionaries in order to enable efficient communication with other Main Nodes and Nodes. Similarly to *Config*, *Graph* implements the *IGraph* interface.

```
public Dictionary<IPAddress, HashSet<EventInternal>> NodeEvents =
                            new Dictionary<IPAddress, HashSet<EventInternal>>();
```

Code Snippet 10: Graph partial properties

The `NodeEvents` Dictionary is used to track which Nodes are allowed to execute which events. When receiving a request to execute an event from a given Node, a Main Node does not have a string identifier of the Node, but it does have their IP. This IP can be used to index `NodeEvents`, and find a HashSet of references to internal events - meaning local events, that the Node has permission to execute.
Graph has more properties than what has been mentioned thus far, but these will be saved for later, when their usage will become clear.

4.3.2.1 CreateEvent()

Most of this method has already been described in subsection 3.3, except the fact that they take an additional parameter in order to give Nodes permission to execute the created *EventInternal*. This parameter is a string array, which should contain the one or more identifier matching the identifier of a Node, located in `Nodes`. A reference to the newly created *EventInternal* is then added to the HashSet of the entries in `NodeEvents`.

```
foreach (string owner in owners)
    {
        IPAddress client = Nodes[owner];
        NodeEvents[client].Add(Events[name]);
    }
```

Code Snippet 11: Code snippet from the end of `CreateEvent()`

4.3.2.2 EventInternal

Most of this class has already been mentioned in subsection 3.3, but in order to allow asynchronous execution of events, we had to add some more properties to this class. First, we needed a way to determine whether a given instantiation is blocked, meaning it is being executed or about to have its markings changed as a result of executing another event. Second, we need to be able to ignore this block if the current execution of the event is the one that blocked it. We accomplish this with a Tuple named `Block`, containing a `boolean` to indicate whether an event is blocked or not, and a `Guid` to indicate which execution blocked the event. We could alternatively have used the default value of a Guid to indicate that the event is not blocked, but we chose to use a boolean because it is faster to determine blocking status by checking a single boolean, rather than an entire Guid every time.
Additionally, in order to ensure that two concurrent executions do not modify the `Block` property at the same time, we added a `Semaphore` as another property.

4.3.2.3 CreateEventExternal()

This method takes 3 strings as parameters: a `name`, `label`, and `peer`. `name` should match the name defined when the event was created on the Main Node which it is located on. This Main Node is defined with the `peer` parameter, which is used to index the previously mentioned `MainNodes` Dictionary in `Config`. These parameters are given to create an instance of *EventExternal*. The *EventExternal* is simply a struct used to hold exactly these 3 parameters. This instance is then added to another Dictionary `Graph.EventsExternal`, which holds all ExternalEvents, indexed by the event name as a string.

4.3.2.4 AddRelationExternal()

This method is almost exactly like `AddRelationInternal`, as described in **Code Snippet** 3. In this method, however, the `eNameTo` string should refer to an *EventExternal* in the `Graph.EventsExternal` Dictionary, allowing us to have relations from local events to events located on other Main Nodes. Additionally, we do not increment any external events' `Condition` or `Milestone` markings.
Similarly to `AddRelationInternal()`, we add a reference to `eTo` in `eFrom`'s HashSet for the given relation. Note that the *EventInternal* class has separate HashSets for referencing relations to Internal and External events.

4.3.2.5 Load()

The `Load()` method is responsible for converting a global DCR graph to a *Graph*. It takes a single parameter: the string path to an .XML file containing all the information associated with the global graph, and it compares this with the instance of *Config* that has already been supplied to the *Graph* instance. By doing this, it figures out which parts of the global DDCR graph should be inserted into the local *Graph* instance.

```
public void Load(string path)
{
    XElement graphXML = XElement.Load(path);
    var eventNameMappingExternal = new Dictionary<string, EventExternal>();
    var eventMarkingMappingExternal = new Dictionary<string, bool[]>();
    ...
    foreach (var ev in events.Elements("labelMapping"))
    {
        var value = ev.Attribute("eventId").Value;
        var mainNode = value.Substring(0, value.IndexOf('_'));
        var label = ev.Attribute("labelId").Value;
        var name = value.Substring(mainNode.Length + 1);
        bool executed = (bool)markings.Elements("executed")
        .Descendants().Attributes("id").Any(x => x.Value == value);
        ...
        if (mainNode == config.Name)
        {
            foreach (KeyValuePair<string, IPAddress> node in config.Nodes)
            {
                foreach (string internalEvent in config.NodeEventsMapping[node.Value])
                {
                    if (internalEvent == name)
```

```
                        CreateEvent(name, label, node.Key.Split(),
                                    included, pending, executed);
                }
            }
        }
        else
        {
            var tempEventExternal = new EventExternal(name, label, mainNode);
            eventNameMappingExternal.Add(name, tempEventExternal);
            eventMarkingMappingExternal.Add(name,
                                new bool[3] { executed, included, pending });
        }
    }
    LoadRelations(relations.Elements("conditions").Elements(),
    Relation.Condition, eventNameMappingExternal, eventMarkingMappingExternal);
    ...
}
```

Code Snippet 12: Partial `Graph.Load` code

First, the XML graph is loaded as an *XElement*, which makes it easy to index the XML file by its native Element and Attribute sections. Temporary mappings between Main Node names and the corresponding `EventExternal` is contained in, and added to the `eventNameMappingExternal` dictionary, and mapping between these names and the marking of the external events is contained in and added to the `eventMarkingMappingExternal` dictionary. This is required, because the `EventExternal` objects do not have any properties related to their marking. The reason why we want to know these markings in this case, is simply to infer the starting markings of our local events. For example, in order to figure out whether a local event's `Condition` or `Milestone` property should be initialised as nonzero, we need to know the markings of these external events which may have a Condition or Milestone relation to our local event. The foreach loop iterates through all events, and determine whether they are local or external events by comparing them to the name defined in the `config`. The event names in the XML file should be formatted in a specific way to allow this, mentioned in section 7.

After all events have been instantiated, the `LoadRelations()` is called for every relation type.

4.3.2.6 LoadRelations()

LoadRelations() gets passed each relation in the global graph as an enumerable XElement, and iterates through them. It is also passed the type of relation it is iterating through, as well as the two temporary dictionaries mentioned in `Load()`.

By comparing the name of the source and target event of each relation with `graph.Events` and the temporary `eventNameMapping` which was created in `Load()` and passed to this method, it can infer whether these events are local or external.

```
private void LoadRelations(IEnumerable<XElement> relations, Relation relationType,
            Dictionary<string, EventExternal> eventNameMapping,
            Dictionary<string, bool[]> eventMarkingMapping)
{
    foreach (var relation in relations)
    {
        var sValue = relation.Attribute("sourceId").Value;
        var sName = sValue.Substring(sValue.IndexOf('_') + 1);
        var tValue = relation.Attribute("targetId").Value;
        var tName = tValue.Substring(tValue.IndexOf('_') + 1);

        //Check if Local -> Local
        if (Events.ContainsKey(sName) && Events.ContainsKey(tName))
        {
            AddRelationInternal(sName, tName, relationType);
        }

        //Check if Local -> External
        else if (Events.ContainsKey(sName))
        {
            var tMainNode = eventNameMapping[tName].Peer;
            var tLabel = eventNameMapping[tName].Label;

            if (!eventsExternal.ContainsKey(tName))
                eventsExternal.Add(tName, new EventExternal(tName, tLabel, tMainNode));

            AddRelationExternal(sName, tName, relationType);
        }

        //Check if External -> Local
        else if (Events.ContainsKey(tName))
        {
            bool[] sMarking = eventMarkingMapping[sName];
            switch (relationType)
            {
                case Relation.Condition:
                    if (!sMarking[0] && sMarking[1])
                        Events[tName].Condition++;
                    break;
                case Relation.Milestone:
                    if (!sMarking[0] && sMarking[2])
                        Events[tName].Milestone++;
                    break;
            }
        }
    }
}
```

Code Snippet 13: Partial `Graph.Load` code

If the source and target event of the relation in the foreach loop are both local, `AddRelation()` can simply be used. If the source is local, and target is external, the temporary `EventExternal` in `eventNameMapping` is made permanent by adding it to the `Graph.EventsExternal` dictionary, and afterwards `AddRelationExternal()` is called. If the source is external and the target is local, we may have to increment the `Condition` or `Milestone` properties of the target event, depending on the marking of the external source event. This is where the temporary initial markings in the `eventMarkingMapping` are necessary. Finally, if both the target and source event are external, they do not affect our graph directly, before any executions, and we are not interested in them.

4.3.2.7 Graph.Execute()

The `Execute()` takes a single parameter: the string name of an event, and returns a boolean to indicate whether execution was successful or not. This name must correspond with a previously added *EventInternal* to the *Graph*, using `Graph.CreateEvent()`.

```
public bool Execute(string eventName)
{
EventInternal ev = Events[eventName];

ev.Semaphore.WaitOne();
if (ev.Block.Item1) //Check if ev is Blocked
{
    ev.Semaphore.Release();
    return false;
}
Guid execId = Guid.NewGuid();
ev.Block = new Tuple<bool, Guid>(true, execId);
ev.Semaphore.Release();

Blocked.Add(execId, new List<EventInternal>());
Blocked[execId].Add(ev);

DateTime execTime = DateTime.Now;
```

Code Snippet 14: Partial Graph.Execute() method

We use the parameter string to index the *EventInternal* in `Graph.Events`. We then lock the events semaphore, and check whether it is currently being blocked with its `Block` property, and if it is, we return false.
Otherwise, we generate a `Guid` for the current execution, and use this to change the events `Block` tuple, in order to indicate that the event is now blocked by the current execution. We then unlock the *EventInternal*'s Semaphore. We also add a new entry to a Dictionary `Graph.Blocked` using the execution ID, and add a reference to the currently executing event to this entry.

```
public Dictionary<Guid, List<EventInternal>> Blocked =
                                    new Dictionary<Guid, List<EventInternal>>();
```

Code Snippet 15: Graph.Blocked property

This Dictionary is used to keep track of which events under which execution is blocked, in order to unblock them when we are done executing the event that causes them to be blocked - indicated by the aforementioned `Guid`.

```
if (!ev.Enabled() || !TryBlockInternal(ev, execId))
{
    UnblockEvents(execId);
    return false;
}
```

Code Snippet 16: Partial Graph.Execute() method, continued

As mentioned in subsection 3.3, we check whether the executing event is enabled. If it is enabled, we attempt to block the other events whose markings will change if the execute is successful, by calling the `TryBlockInternal()` method, which will try to block all events that may be affected by the execution. If any of these events are already blocked by another execution, `TryBlockInternal()` will instantly return false, and we call the `UnblockEvents()` method which unblocks all events, if any, that were blocked by `TryBlockInternal()`, as well as the event we are trying to execute. Unblocking is easy because all the blocked events were blocked using the current execution ID, and can easily be indexed that way. `Execute()` will then return false. If no events were already blocked, however, they are now blocked using our execution ID and we may safely proceed.
We then generate the messages, if any, that need to be sent to other Main Nodes in case the executing event has external relations. This is done by the `GetPeerMessages()` method, and is detailed in paragraph 4.3.2.9.

```
Dictionary<string, string> peerMessages = GetPeerMessages(ev);
if (peerMessages.Count != 0 && !TryBlockExternal(peerMessages, execId,
                                    execTime.ToString(Config.Culture)))
{
    UnblockEvents(execId);
    return false;
}
```

Code Snippet 17: Partial Graph.Execute() method, continued

If it turns out that we have any messages that needs to be sent, we try to block the external events by sending a block message using `TryBlockExternal()`. `peerMessages` is here passed to `TryBlockExternal`, as it contains information about which Main Nodes and what messages should be sent, in order to inform the Main Node what events should be blocked.

```
bool externalResult = TryExecuteExternal(tasksEmpty, execId);
if (externalResult)
{
    UpdateMarkingsInternal(ev);
    ev.Executed = true;
    ev.Pending = false;

    Log.Add(execTime, ev.Name);
    Dictionary<string, Task<string>> tasksUnblock =
            MessagePeers("UNBLOCK", execId, tasksEmpty);
    ...
```

Code Snippet 18: Partial Graph.Execute() method, continued

If all external events were blocked successfully, we try to change their markings, using the `TryExecuteExternal()` method. Once we have confirmed all external markings have been changed successfully, we, like in subsection 3.3, change the local markings of affected events with `UpdateMarkingsInternal()`, and change the markings of the executed event to indicate that it has been executed and is no longer pending. Additionally, we add this execution to another property of *Graph*, a local log of executed events and their execution time. The *Log* class is detailed more in depth in paragraph 4.3.2.13.
Then, regardless of whether `TryExecuteExternal()` was successful or not, we unblock the external events with the `MessagePeers()` method. This is because, if `TryExecuteExternal()` was not successful, it will automatically send a "REVERT" message, reverting the marking of the executed events.

```
    ...
        IEnumerable<string> deadPeers =
            from x in tasksEmpty.Keys.Except(tasksUnblock.Keys) select x;
        foreach (string deadPeer in deadPeers)
        {
            TryUnblockPeerAsync(deadPeer, execId.ToString());
        }
    }
    UnblockEvents(execId);
    return externalResult;
}
```

Code Snippet 19: Partial Graph.Execute() method, continued

Here, we ensure that all Main Nodes successfully received the "UNBLOCK" message, by comparing replies to sent messages. If a Main Node did not reply, we will repeatedly try to contact it, telling it to unblock the event. This is done asynchronously with the method `TryUnblockPeerAsync`, therefore it won't affect the current execution. Finally, we unblock all our internal events with `UnblockEvents()`, and return a boolean indicating whether or not the execution was successful.

4.3.2.8 LinkedDictionary

The *LinkedDictionary class* is a new datatype we have created for building up peer messages. The class consists of two dictionaries, where the first Dictionary's values are linked to the second Dictionary's keys. Both keys are strings, and both values are HashSets of strings. An example of how this works is showcased when using this class in the `GetPeerMessages()` method.

4.3.2.9 GetPeerMessages()

The `GetPeerMessages()` method is used to create the execution related messages sent to other Main Nodes. The method takes an `Event` as input, which is an abstract class inherited by *EventInternal* and *EventForeign*, as this allows `GetPeerMessages()` to be called with either class. The method can be split into two parts, the filling of a *LinkedDictionary* and the filling of a Dictionary with strings as keys and values, called `messages`. In `messages` the keys are the external Main Nodes' names and the values are the message which is sent to the Main Nodes as seen in Figure 9.

`GetPeerMessages()` starts out by checking if the input event *ev* is an *EventInternal* or *EventForeign*. If it is an *EventInternal*, that means that the event can have external relations and we therefore have to account for these. The method then checks all external relations for *ev* and adds them to the *LinkedDictionary* if any are found. There are two interesting cases, which are the event's `ConditionsExternal` and `MilestonesExternal` relations, since these depend on *ev*'s own markings before being added. In **Code Snippet** 20 we only add a condition message if *ev* has not already been executed. If *ev* has already been executed, then we know from subsubsection 3.2.2 that the event, which *ev*'s Condition relation is towards, is enabled. We also know from this `executed` marking that at some point prior to this, when *ev* was executed the first time, a message was sent, decrementing the `Condtion` marking by 1. Like the condition relation, we know from subsubsection 3.2.2, that we only add a milestone relation to the message is if the input event is pending. If the event is not pending the milestone relation is dormant and we do not have to inform the affected event.

```
if (!evI.Executed)
{
    foreach (EventExternal c in evI.ConditionsExternal)
        peerMessages.Add(c.Peer, c.Name, "condition-");
}

if (ev.Pending)
{
    foreach (EventExternal m in evI.MilestonesExternal)
        peerMessages.Add(m.Peer, m.Name, "milestone-");
}
```

Code Snippet 20: Code snippet the: ConditionsExternal and MilestoneExternal cases

Next, the method looks at *ev*'s internal Response, Includes and Exclude relations. We omit Condition and Milestone since they would already have been set by the `load()` method at initialisation, or the event started out as excluded and in that case no propagation has occurred since the relations are currently inactive. Firstly, we look a the Response relation between the input event *ev* and the targeted event e_2. If e_2 is included and is not currently pending, we look at e_2's external Milestone relations to e_3 and if there is such a relation, then we add a message to e_3 to the *LinkedDictionary*, telling it to increment its `Milestone` property by 1. The code is shown in **Code Snippet** 21.

```
foreach (EventInternal e2 in ev.Responses)
{
    if (e2.Included && !e2.Pending)
    {
        foreach (EventExternal e3e in e2.MilestonesExternal)
            peerMessages.Add(e3e.Peer, e3e.Name, "milestone+");
    }
}
```

Code Snippet 21: Code snippet of the internal response relations

It then continues on to the internal Include relations between e and e_2, here we only add messages if e_2 is not already included. If e_2 is included, then an Include relation will not have any effect on the markings and nothing changes. After we check if e_2 is included, we perform the same operation as in **Code Snippet** 20, but this time we check e_2's external relations instead of ev's internal relations, if any are found.

We then look at the Exclude relations between ev and e_2, and check if e_2 is included. If it is not, we do not send anything as excluding something that is already excluded does not change any markings. We also check if ev is not the the same event as e_2, i.e we check, that it does not exclude itself. If it excludes itself, we do not add anything to the *LinkedDictionary.* This is because, if ev were blocking an external event with a Condition or Milestone relation, executing ev will result in these relations no longer being active, regardless of ev being excluded or not - as seen in **Code Snippet** 20. If both of these checks are true, we perform the same operations as in **Code Snippet** 20 except we check e_2's external relations, instead of ev. The code is shown in **Code Snippet** 22

```
foreach (EventInternal e2 in ev.Excludes)
{
    if (e2.Included && ev != e2)
    {
        if (!e2.Executed)
        {
            foreach (EventExternal e3e in e2.ConditionsExternal)
                peerMessages.Add(e3e.Peer, e3e.Name, "condition-");
        }

        if (e2.Pending)
        {
            foreach (EventExternal e3e in e2.MilestonesExternal)
                peerMessages.Add(e3e.Peer, e3e.Name, "milestone-");
        }
    }
}
```

Code Snippet 22: Code snippet of the internal exclude relations

In order to build up the string messages for updating markings of events located on external Main Nodes, we want to minimize message overhead by sending only one message to each peer. We do this by using the aforementioned *LinkedDictionary* class. In `GetPeerMessages()`, we might build up a *LinkedDictionary* with the following queries, albeit automatically and not manually as shown below.

```
var peerMessages = new LinkedDictionary();
peerMessages.Add("peer1", "event1", "response");
peerMessages.Add("peer1", "event1", "condition");
peerMessages.Add("peer1", "event2", "exclude");
peerMessages.Add("peer3", "event6", "include");
```

Code Snippet 23: Example usage of LinkedDictionary. Peers refer to external Main Nodes. Note that we assume that no events on different Main Nodes have identical names.

These queries will end up making the structure of the *LinkedDictionary* look like this:

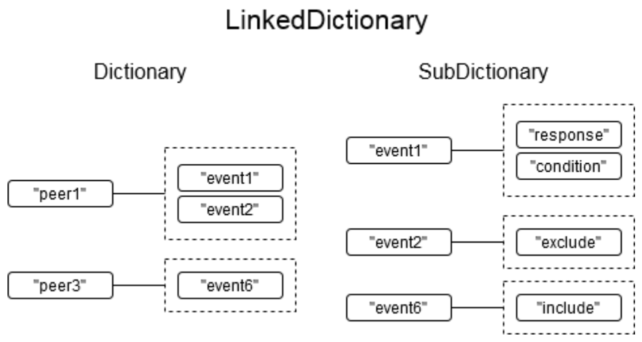


Figure 9: peerMessages after **Code Snippet** 23. The HashSets are indicated by dotted boxes.

After building the *LinkedDictionary*, the method converts it into a regular Dictionary with strings as both a keys and values. The key is the name of the Main Node who should receive the message, and the value is the message itself - containing all the events located on that Main Node, along with which relation is affecting them. Based on this relation, the receiving Main Node can infer how its marking should change. The example in **Code Snippet** 23 and Figure 9 produces the Dictionary:

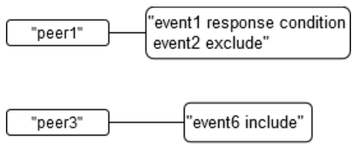


Figure 10: Example return Dictionary for GetPeerMessages()

This Dictionary is then returned by `GetPeerMessages()`.

4.3.2.10 MessagePeers()

`MessagePeers()` is a method in *Graph* responsible for for sending messages of a specific format. It is used when sending "EXECUTE", "REVERT" and "UNBLOCK" messages, since they are structured in the same way but with a different keyword, see subsubsection 4.3.3. The method takes a string, called `command`, representing which kind of message is being sent, a `Guid` associated with an execution, a Dictionary called `tasksPrev`, and finally a boolean `lastMsg` indicating whether the sent message is the last in a sequence. The keys in `tasksPrev` are Main Node names and the values are `Task<string>`, representing replies to a previously sent message to the Main Node, if any have been sent. The method starts by creating a new Dictionary with same type as `tasksPrev`, called `tasks`. The method then checks for each Main Node in `tasksPrev` if the result of its message is "UNAVAILABLE". If it is not, we add the Main Node name as key in `tasks`, and a `SendAsync()` as value, using the `command` and execution id as parameter to the method. The `SendAsync()` method is detailed in paragraph 4.3.4.1.
The method then waits for all the Tasks in `tasks` to complete, before returning the `tasks` Dictionary. The code is shown in **Code Snippet** 24.

```
public Dictionary<string, Task<string>> MessagePeers(string command,
    Guid executeId, Dictionary<string, Task<string>> tasksPrev, bool lastMsg)
{
    Dictionary<string, Task<string>> tasks = new Dictionary<string, Task<string>>();

    foreach (string peerName in tasksPrev.Keys)
    {
        if (tasksPrev[peerName].Result != "UNAVAILABLE")
        {
            tasks[peerName] = MainNodes[peerName]
                .SendAsync(string.Format("{0} {1}", command, executeId), lastMsg);
        }
    }

    Task.WaitAll(tasks.Values.ToArray());
    return tasks;
}
```

Code Snippet 24: `MessagePeers()` method

4.3.2.11 TryBlockExternal()

`TryBlockExternal()` is a method in *Graph* which sends a specific type of message, namely "BLOCK". It does so, much like `MessagePeers()`, but has a more complex message. Unlike `MessagePeers()`, it checks the replies from the Main Nodes before returning and returns a bool. The method takes a Dictionary called `peerMessages` constructed by `GetPeerMessages()` as parameter, a `Guid` associated with the execution, and an execution time as a string. The method sends messages by creating a `tasks` Dictionary, like `MessagePeers()` does. It then proceeds to send a "BLOCK" message to each Main Node in `peerMessages` and waits for the result. After all the Main Nodes have replied, `CheckPeerReplies()` is called to check if any Main Node responded with "UNAVAILABLE". If so, it calls `MessagePeers()` in order to unblock every Main Node that replied with "SUCCESS", and lastly returns `false`. If every Main Node replied with a "SUCCESS", then the blocking of external Main Nodes were successful and it returns `true`.

4.3.2.12 Revert

In order to handle a failed execution, like we see in Figure 8, we have designed a way to ensure a global graph's validity by allowing Main Nodes to revert markings. The need for reverting markings occurs when an execution fails, e.g. by the server hosting the Main Node crashing or it returning "UNAVAILABLE". When this happens a "REVERT" message is sent to the Main Node that replied that their execution was successful. In order to enable this reversibility, we use the two methods `GetNewMarkingsReversible()` and `ChangeMarkings()`. The first creates the list of tuples where each item in the list represents an `action`, and the second method uses these `actions` to change the markings. These markings can be changed safely, because the events are blocked. When the `actions` are created by `GetNewMarkingsReversible()`, they are saved in a foreign event, as explained in paragraph 4.3.3.1. `GetNewMarkingsReversible()` takes an *EventForeign* as input which enables it to read the *EventForeign*'s markings and fill its actions list. `GetNewMarkingsReversible()` builds the list of actions by looking at the internal relations of the input *ForeignEvent* e and for each relation it checks, similarly to `GetPeerMessages()` in paragraph 4.3.2.9, the current markings of the targeted event e_2. If an `action` is added to the list, the added tuple consists of *EventInternal*, a *Relation* enum, and a bool. The *EventInternal* indicates the targeted event by the relation, the *Relation* enum is the type of the relation, and the bool indicates how the marking should be changed. For example, if the `action` is (e_3, `Relation.Exclude`, `false`), then e_3's `Included` property should be set to `false`. If it however is (e_3, `Relation.Condition`, `false`), then e_3's `Condition` property should be decremented by 1. When the list of `actions` has been built, it is added as a property of e. e can then be passed as parameter to `ChangeMarkings()`, which iterates through the list and changes the markings accordingly. `ChangeMarkings()` also has a bool as parameter, whose value determines whether the markings should be reversed or not. If the bool is set to true then all the bools in the list of actions are negated and if the input bool is set to false - then they remain the same. This way, if a execution goes wrong, we can return to Main Nodes and ask them to revert their changes by finding the *EventForeign* corresponding to the execution ID and calling `ChangeMarkings()` with the event and a bool set to `true`.

The need to reverse originated from the desire to keep the global graph synchronised regardless of failed executions. Without it, some Main Nodes might become desynchronised by returning "UNAVAILABLE" to a execution request, while the remaining Main Nodes change their markings. This way, we ensure that all the affected Main Nodes complete their marking changes before concluding that the execution was a success.

4.3.2.13 Log

The `Log` class implements the *ILog* interface and inherits from a list of tuples, where the first item in the tuple is a `DateTime` and the second item is a string representing an event name. Because of this inheritance, the `Log` contains when events have been executed on a Main Node. Additionally, when sending logs in the network, we need a reliable way to convert them to and from strings. *Log* therefore has an override of the method `ToString()`, which converts a non-empty log into a single string. This string can then be sent to other Main Nodes and converted back into a log again.

In order to request a global log, we use the `GetGlobal()` method, which takes three parameters: First, a Dictionary with Main Node names as key, indicating who to send requests to, and the given Main Node's *IClient* instance. Second, a string containing the names of previous Main Nodes that have already been sent a request, and third, a `Guid` unique for the current log request.

```
public ILog GetGlobal(Dictionary<string, IClient> peers, string prevPeers, Guid id)
{
    List<Task<string>> tasks = new List<Task<string>>();
    string peerNames = prevPeers;
    foreach (string peer in peers.Keys)
        peerNames += " " + peer;

    foreach (IClient client in peers.Values)
        tasks.Add(client.SendAsync(string.Format("LOG {0}\n{1}",
                                    id.ToString(), peerNames), true));

    Task.WaitAll(tasks.ToArray());
    Log globalLog = new Log(this, config);
    foreach (Task<string> task in tasks)
    {
        string[] line = task.Result.Split('\n');
        if (line[0] == "LOG IGNORE" || line[0] == "LOG EMPTY")
            continue;
        if (line[0] == "UNAVAILABLE")
            return null;
        foreach (string entry in line)
        {
            string[] curLine = entry.Split(' ');
            /*The method DateTime.ToString() converts to a string with a space,
              which are separated with String.Split. We recombine them and parse
              them back to the DateTime class below. */

            DateTime time = DateTime.Parse(string.Format("{0} {1}",
                        curLine[0], curLine[1]), config.Culture);
            //curLine[2] should be event name of the executed event.
            globalLog.Add((time, curLine[2]));
        }
    }
    return globalLog;
}
```

Initially, the string containing the names of the previous Main Nodes only contains the name of the Main Node that sends the first request. The method then appends all known Main Nodes to this string, as these will all be sent a request. Each Main Node that receives one of these requests will do this, although they will not append any duplicate names. This string is sent with the log request so that Main Nodes receiving this message can see which other Main Nodes have already received the request, and not send them another. A message is then constructed for each Main Node that has not already received a message. This message contains the Log request, the aforementioned `Guid`, and the string of previous Main Nodes. Notably, a boolean `true` is as second parameter to `Client.SendAsync()`, indicating that the message is the last (and only) message, meaning an "END OF TRANSMISSION" character should be appended to the message.
When a Main Node receives the `Guid`, it checks if it has already received it and if so, it ignores the request. See paragraph 4.3.3.4 for more details on how a received log request is handled.
When the method receives replies to all of its outgoing log requests, it converts the string into log entries, and adds them to the log. It will skip over a reply if it is "LOG EMPTY" or "LOG IGNORE", indicating that a Main Node did not have anything in its log and a Main Node had already received the request, respectively. When all the entries have been added or skipped over - the log is returned. Each local log is a trace associated with a run on the local graph, like we see in subsubsection 3.2.4, and the global log is a trace associated with a run on the global graph.

4.3.2.14 Accepting

The way we check if the global graph is Accepting is separated into two methods:
`GetAcceptingInternal()`, as seen in **Code Snippet** 6, and `GetAcceptingExternal()`, where the first method checks the local graph and the second method queries other Main Nodes for their state. `GetAcceptingExternal()` propagates the Accepting request to other Main Nodes much like `Log` does in paragraph 4.3.2.13. Before `GetAcceptingExternal()` sends out any request, it checks its own state to see if it is Accepting with `GetAcceptingInternal()`, and if it returns `false`, we know that there is no need to send requests to other Main Nodes, since the global state cannot be `true`. If `GetAcceptingInternal()` returns `true`, the Main Node will propagate the request like `Log` and if it receives any reply that says "FALSE", it then returns `false`. Otherwise, the global accepting state is true and it returns `true`, unless there is a connection error, in which case it will throw an exception, which is handled in `graph.ParseNode()`.

4.3.3 Parser

As mentioned in subsubsection 4.2.1, the parser interprets the different received messages. Each message follows a specific pattern and can therefore be parsed in the same way. Some of these messages are appended with a terminator character, indicating that it is the last message and that the receiver should close the socket after receiving the message.
In this section we have chosen to explain the 4 most important branches in the `ParseMainNode()`, which is "BLOCK", "EXECUTE", "REVERSE", and "LOG". "LOG" and "ACCEPTING" requests are very similar, and ACCEPTING is therefore omitted.

At the start of `ParseMainNode()`, the method starts by splitting the received message by newlines, and storing them in a string array. The first entry in the array is then again split by white spaces and stored in another string array. (Note that arrays are 0-indexed in C#). The first element in that string array is then used to match in a switch statement, and is the keyword for what kind of message has been received. We can see the splitting of messages in **Code Snippet** 25

```
string[] newLineMessage = message.Split('\n');
string[] splitMessage = newLineMessage[0].Split(' ');
```

Code Snippet 25: Splitting of received messages

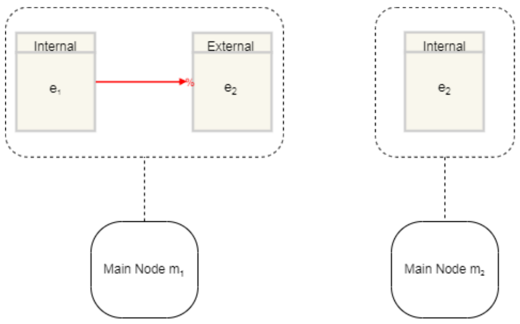
The different messages from Main Nodes follow strict structure, as seen below:

- BLOCK <ID> <TimeDate>
  «Event e_1> <Relation r_1, Relation r_2, ... , Relation r_n»
  $\vdots$
  «Event e_n> «Relation r_1, Relation r_2, ... , Relation r_n»
- EXECUTE <ID>
- REVERT <ID>
- UNBLOCK <ID>
- ACCEPTING <ID>
  <previous peers> <known peers>
- LOG <ID>
  <previous peers> <known peers>

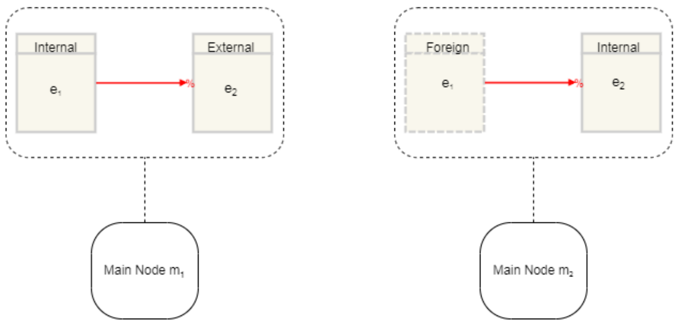
4.3.3.1 EventForeign

EventForeign is an object, that is a copy of e_1 from m_1 in m_2, used to save relations and marking changes in another Main Node. The event is created in the BLOCK case in the `ParseMainNode()` method and is only used for the duration of the execution before being discarded and collected by a garbage collector. We can see the *EventForeign*'s life cycle in Figure 11, where it is created when m_2 receives a BLOCK message from m_1, and its relation to the *EventInternal* e_2 in m_2 is created. This relation is used in `GetMarkingsReversible()` to create `actions`, symbolising the changes in markings, which is saved in the *EventForeign*. When the *EventForeign* is created, it is added to a Dictionary called `EventsForeign` which has an execution ID as key and an *EventForeign* as value. This provides an easy way to locate the *EventForeign* and serves as the only reference to the object. When an "EXECUTE" message is received, the *EventForeign* is found in this Dictionary, and used by the parser to complete the execution by changing the markings. When this either fails or is successful, the *EventForeign* is removed from the Dictionary and is at some point soon after, collected by a garbage collector since the only reference to the object is gone.

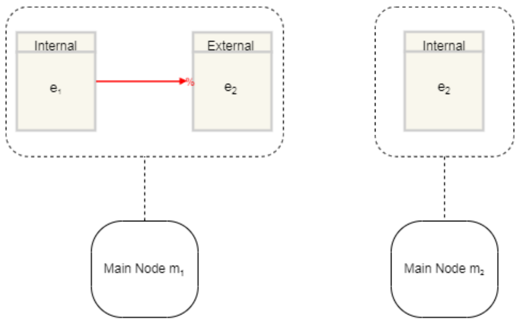
The *EventForeign* serves two purposes: to preserve which markings should change between the "BLOCK" and "EXECUTE" messages with its `actions` property, and to access the external events of e_2 in order to find out which Main Nodes it needs to propagate the changing of markings to. An important note to make is that the *EventForeign* is never executed but simply passes its `actions` property to the `ChangeMarkings()` method.



(a) Before an execution attempt



(b) After BLOCK message and during execution



(c) After the UNBLOCK message following a successful execution

Figure 11: Example of how a foreign event is made and disposed.

4.3.3.2 Case: Block

The Block case starts out by finding the execution ID and the execution time in the split message string array. It then proceeds to identify which of the local events are contained in the message by looping through the `newLineMessage` array and matching the first element in the newline with the events contained in the local graph. If the graph contains the element, its name is added to a string array called `events`. The code is shown in **Code Snippet** 26.

```
Guid execId = Guid.Parse(splitMessage[1]);
string execTime =
    splitMessage[2] + " " + splitMessage[3];
```

```
string[] events = new string[newLineMessage.Length - 1];
for (int i = 1; i < newLineMessage.Length; i++)
{
    string[] line = newLineMessage[i].Split(' ');
    if (graph.Events.ContainsKey(line[0]))
        events[i - 1] = line[0];
    else
        return "UNAVAILABLE";
}
```

Code Snippet 26: Initial steps in the BLOCK case

Block then continues to create a list of `actions` as described in paragraph 4.3.3.1. Afterwards a *EventForeign* is created. We then loop through each line in the message, and add relations between the foreign event and local events. These relations are then used to populate `actions`. We can see this in **Code Snippet** 27. Note that the switch inside the loop has been shortened to preserve space.

```
List<(EventInternal, Relation, bool)> actionsTemp =
    new List<(EventInternal, Relation, bool)>();

EventForeign e = new EventForeign();
for (int i = 1; i < newLineMessage.Length; i++)
{
    string[] line = newLineMessage[i].Split(' ');
    EventInternal ev = graph.Events[line[0]];
    for (int j = 1; j < line.Length; j++)
    {
        switch (line[j])
        {
            case "response":
                e.Responses.Add(ev);
                break;

            case "milestone+":
                e.Pending = true;
                actionsTemp.Add((ev, Relation.Milestone, true));
                break;

            case "milestone-":
                actionsTemp.Add((ev, Relation.Milestone, false));
                break;
        }
    }
}
```

Code Snippet 27: Looping through newlines in BLOCK

After the loop, we add the *EventForeign* to the `EventsForeign` dictionary in `Graph` with the execution ID as key and the *EventForeign* as value. Then, if it does not already exist, we add the execution ID to the `Blocked` Dictionary in `Graph` with the execution ID as key and a list of *EventInternal* as value. We then proceed to try and block the internal events in `TryBlockInternal()`, which takes an event and an execution ID. If this fails, we unblock the already blocked events and break out of the BLOCK case. If it was successful, we retrieve the Main Node messages from `GetPeerMessages()` in *Graph* and save it in `peerMessages`. When that is done, we attempt to block all the external events found in `peerMessages` with the execution time and ID - if this fails we unblock the events we have blocked and break out of the "BLOCK" case. We can see this in **Code Snippet** 28.

```
graph.EventsForeign.Add(execId, e);
if (!graph.Blocked.ContainsKey(execId))
    graph.Blocked.Add(execId, new List<EventInternal>());

if (graph.CheckBlocked(events) && !graph.TryBlockInternal(e, execId))
{
    graph.UnblockEvents(execId);
    break;
}

Dictionary<string, string> peerMessages = graph.GetPeerMessages(e);
if (!graph.TryBlockExternal(peerMessages, execId, execTime))
{
    graph.UnblockEvents(execId);
    break;
}
```

Code Snippet 28: Blocking internal and external events in BLOCK

When all the blocking is successful, we save the `actions` to be done and the Main Nodes to propagate to in the foreign event *e*. We use the `GetNewMarkingsReversible()` method to generate reversible `actions` for when the event is executed, in case anything goes wrong. These `actions` are combined with the ones in **Code Snippet** 27 and saved in the foreign event. When the Main Nodes to propagate to and `actions` have been saved we return a "SUCCESS" to the blocking request.

4.3.3.3 Case: Execute

When the parser receives an "EXECUTE" message and enters the "EXECUTE" branch, it starts out by re-creating the execution ID from the `splitMessage`. Afterwards it checks if the `EventsForeign` contains the execution ID as a key - if it doesn't it jumps out of the switch statement. However, if it does contain the key, then we wish to change the markings dictated by the foreign event and propagate the execute message. To change markings, we first locate the *EventForeign* in the dictionary and then we call the `TryExecuteExternal()` method, which takes two parameters: a Dictionary with a string as key and a Task<string> as value, and an execution

ID. This Dictionary can be found in the *EventForeign* as a property, and is created and saved during the initial BLOCK message. If `TryExecuteExternal()` fails, i.e returns `false`, we unblock the blocked events and remove the *EventForeign* from the `EventsForeign` Dictionary. If it succeeds, we update the local markings via the `ChangeMarkings()` method, which takes an *EventForeign* and a boolean that indicates whether the markings should be reversed or not. If the boolean is false then `ChangeMarkings()` should not reverse.

The groundwork for the propagation is laid in the BLOCK case of the parser, where the Main Nodes to propagate to is saved in the *EventForeign* object, along with the actions for changing markings. We propagate to the Main Nodes before we change the markings in the current Main Node. This way, we make sure that all the propagation is successful before we change the markings, and when the local markings are changed, it will always be successful since the affected events are blocked. The code is shown in **Code Snippet** 29.

```
case "EXECUTE":
    {
        Guid execId = Guid.Parse(splitMessage[1]);
        if (graph.EventsForeign.ContainsKey(execId))
        {
            EventForeign foreignEvent = graph.EventsForeign[execId];

            bool propagateResult =
                graph.TryExecuteExternal(foreignEvent.peersPropagate, execId);

            if (!propagateResult)
            {
                graph.UnblockEvents(execId);
                graph.EventsForeign.Remove(execId);
            }
            else
            {
                graph.ChangeMarkings(foreignEvent, false);
                return SUCCESS;
            }
        }
        break;
    }
```

Code Snippet 29: The EXECUTE case of the parser.

4.3.3.4 Case: Log

The "LOG" case is very similar to the "ACCEPTING" case, the only difference is what kind of messages they propagate and return. When the LOG case is entered, it re-creates the message id from `splitMessage` and then checks if the list `GetLogIds` contains the ID already. If the list does contain it, it means that a log request that originated from the same Main Node has already reached this Main Node, and we can safely ignore the request. We do so by returning "LOG IGNORE". Next, we add the ID to the list if it did not contain it. We then start the process of propagating the log request to the known Main Nodes. As mentioned in paragraph 4.3.2.13, every "LOG" message is accompanied with a list of names of Main Node, that the message has already been sent to, and the Main Node will add its known Main Nodes to that list if they are not already present. After the unique Main Nodes have been identified, we pass the Dictionary as parameter to the `GetGlobal()` method in the *Log* object, together with a list of previous Main Nodes and the request ID. `GetGlobal()` returns the global log, which is then sorted with `SortLog()`, converted to a string, and sent back to the Node that requested the log. The code is shown in **Code Snippet** 30.

```
case "LOG":
    {
        Guid id = Guid.Parse(splitMessage[1]);
        if (graph.GetLogIds.Contains(id))
            return "LOG IGNORE" + TERMINATOR;
        graph.GetLogIds.Add(id);
        string[] receivedPeers = newLineMessage[1].Split(' ');
        Dictionary<string, IClient> uniquePeers = new Dictionary<string, IClient>();

        foreach (string peerName in config.MainNodes.Keys)
        {
            if (!receivedPeers.Contains(peerName))
                uniquePeers[peerName] = config.MainNodes[peerName];
        }

        string prevPeers = newLineMessage[1];

        ILog reply = graph.Log.GetGlobal(uniquePeers, prevPeers, id);

        return reply.ToString() + TERMINATOR;
    }
```

Code Snippet 30: The LOG case of the parser.

4.3.3.5 Case: Revert

Like many other cases in the `ParseMainNode()`, the "REVERT" case starts off by finding the execution ID from `splitMessage` and then checking if it is contained in the `EventsForeign` Dictionary. It then propagates the revert message to the Main Nodes in the *EventForeign*'s `peerPropagate` field. After propagation, it reverts the markings associated with the execution ID and it does so by calling `ChangeMarking()` with the foreign event and a boolean which is set to `true`. When the changing of markings is done, it unblocks the events and removes the foreign event before it returns "SUCCESS" plus the terminator, indicating that the revert was a success and communication is over. The code is reminiscent of the "EXECUTE" case and is therefore omitted.

4.3.4 Networking

As mentioned previously in subsection 4.2, communication is implemented using two classes - *Client* for sending requests, and *Listener* for receiving requests.

4.3.4.1 Client

The *Client* class is an implementation of *IClient* interface, which only has a single public method. It does, however, also has several useful properties, warranting its own class. Each instantiation of *Client* is meant to represent a Main Node, and thus have properties referring to the Main Node's IP Address and port. These must be given as parameters upon instantiating a *Client*. Clients also have a Semaphore, the purpose of which is explained in the SendAsync() method.
Additionally, a *Client* has an instance of the C# `TcpClient` class as a field, in order to allow us to use the `TcpClient`'s methods - leading us to the next method: `SendAsync`

```
public async Task<string> SendAsync(string msg, bool endConnection)
{
    string reply = string.Empty;
    Semaphore.WaitOne();
    try
    {
        if (!client.Client.Connected && !IsConnected())
            await client.ConnectAsync(ip, port);

        NetworkStream stream = client.GetStream();

        //Prepend M to indicate it is a main node request.
        if (endConnection)
            msg += TERMINATOR;

        byte[] buffer = Encoding.UTF8.GetBytes("M:"+msg);
        await stream.WriteAsync(buffer, 0, buffer.Length);
        buffer = new byte[client.ReceiveBufferSize];
        int replySize = await stream.ReadAsync(buffer, 0, buffer.Length);
        reply = Parser.DecodeMessage(buffer);

        if (reply[reply.Length - 1] == TERMINATOR)
        {
            endConnection = true;
            reply = reply.Substring(0, reply.Length - 1);
        }

    }
    catch (Exception e)
    {
        endConnection = true;
        reply = "UNAVAILABLE";
    }
    if (endConnection)
    {
        try
        {
            client.GetStream().Close();
            client.Close();
        } catch { }
        client = new TcpClient()
        {
            ReceiveTimeout = config.ClientTimeoutMs,
            SendTimeout = config.ClientTimeoutMs
        };
    }
    Semaphore.Release();
    return reply;
}
```

Code Snippet 31: Client.SendAsync() method

We decided to make messaging asynchronous, because it is much more efficient to message potentially many Main Nodes asynchronously, and then wait for a reply from all of them, rather than sending, waiting, repeat. This does, however, bring some potential problems - but these are fixed by using the aforementioned semaphore. Because while we want to send messages to different Main Nodes, using different instantiations of *Client*, at the same time - we do not wish to send messages asynchronously to the same Main Node at the same time, as we want to reuse sockets, reducing the overhead of creating a new socket. Because each Main Node is a `Client` instance, we prevent this by locking the instance's semaphore while we are sending a message - and receiving a reply - with SendAsync(). When we are done, we release the semaphore.
We then check whether our `TcpClient`'s socket is still connected with its .Connected property. Because this property is only updated upon reads or writes however, it may therefore say we are still connected even though we are not. We therefore ensure correctness by calling `IsConnected()`. This method checks whether we already have an existing, open connection to the client - which will be the case if we have recently communicated with it. This is done by looking for the correct IP/port combination in the local computer's currently active TCP connections, and returning `true` if it finds one that is already established - and otherwise false.
Both these checks are needed, because we chose not to automatically dispose the TcpClient socket whenever we are done sending and receiving in the method. This choice meant, that `client.Connected` may sometimes return true, even if the connection is dead. We decided not to close the connection, to reduce the overhead of manually opening a new connection every time we send a message by using existing ones, which is often the case due to the aforementioned protocol of sending a "BLOCK", followed by "EXECUTE", etc. While `IsConnected()` is not fast, we judged it to be faster than creating a new socket and establishing a new TCP connection.
After we have either checked that we already have a valid connection, or made one, we get the

stream used to write and read data. We then append a terminator character to our message, if the second parameter `endConnection` is true. We convert our message to a byte array, and send it. When we are done, we wait for a reply, and convert the reply from byte array to UTF-8 using the `DecodeMessage()` method. We then check if the reply contained a terminator character, and set `endConnection` to true. If `endConnection` is true, we close and therefore dispose of the socket, and create a new one to be used next time communication with the *Client* is required. As mentioned in subsubsection 4.1.1, this prevents us from having to wait for the connection to actually disconnect the TCP connection. We then release the semaphore and return the reply, excluding the terminator (if any), as a string. Note that all of this, is inside a try-catch block. This is because connecting, sending and receiving may throw an exception if the Main Node is unavailable, or does not respond after a given time interval, defined in the *Config* class Exceptions are handled by simply setting endConnect to true, which will close and discard the socket that threw the exception, creating a new one. Exceptions will also set the `reply` variable to "UNAVAILABLE".

4.3.4.2 Listener

The *Listener* class is implemented with two methods and a constructor. The constructor takes two arguments, a *Parser* and an *IConfig*. The *IConfig* contains the relevant network information for the listener, i.e which port to listen on. In **Code Snippet** 32 we see the *Listen()* method, whose responsibility is to accept new TCP connections on the specified port. The method enters into an infinite loop and creates a new `TcpClient` for any incoming connection and passes it on to another method, `HandleConnection`. This infinite loop is the reason that the `Listener` object is on a separate thread, since the loop would block any execution on the main thread. A new `Task` is created to handle the `HandleConnection()` method, since it too contains a potentially infinite loop.

```
public void Listen()
{
    TcpListener listener = new TcpListener(IPAddress.Any, port);
    listener.Start();
    while (true)
    {
        TcpClient client = listener.AcceptTcpClient();
        Task.Run(() => HandleConnection(client));
    }
}
```

Code Snippet 32: *Listen()* method in the `Listener` class

In **Code Snippet** 34 we see the *HandleConnection()* method, which takes a `TcpClient` as parameter. The method's responsibility is to handle a connecting TcpClient and communicate with it. When the method is called, it gets the NetworkStream from the client, and enters an "infinite" while-loop, where it continuously waits for data to be sent from the client.

```
private void HandleConnection(TcpClient client)
```

```
{
    client.ReceiveTimeout = config.ClientTimeoutMs;
    client.SendTimeout = config.ClientTimeoutMs;

    IPEndPoint clientInfo = (IPEndPoint)client.Client.RemoteEndPoint;
    NetworkStream stream = client.GetStream();
    char peerType = ' ';
    bool endConnection = false;
    while (true)
    {
        try
        {
            if (stream.DataAvailable)
            {
            ...
```

Code Snippet 33: First part of the `HandleConnection()` method

This data is converted to UTF-8 and passed to either *ParseMainNode* or *ParseNode* based on the `peerType` and when the called parse method returns, the return value is converted to bytes and sent back to the client. If the client breaks the connection, the infinite loop is broken and the method exits. The message sending and handling is inside an try-catch block, handling any exceptions should they occur, and closing the connection and exiting the loop if an exception is thrown, or a terminator is received from the client. At the end of the method the Task is set to wait for 15 milliseconds before resuming the while-loop. This is done in order to save CPU resources, as the while-loop would otherwise loop as fast as it possibly can.

```
    ...
            byte[] buffer = new byte[client.ReceiveBufferSize];
            int size = stream.Read(buffer);

            string msg = Parser.DecodeMessage(buffer);
            peerType = Convert.ToChar(msg.Substring(0, 1));
            if (msg[msg.Length-1] == TERMINATOR)
            {
                endConnection = true;
                msg = msg.Substring(0, msg.Length - 1);
            }
            string actualMsg = msg.Substring(2);
            string reply = "INVALID";

            if (peerType == 'M')
                reply = parser.ParseMainNode(actualMsg);
            else if (peerType == 'N')
                reply = parser.ParseNode(actualMsg, clientInfo.Address);

            stream.Write(Encoding.UTF8.GetBytes(reply), 0, reply.Length);
        }
    }
    catch (Exception e)
    {
        endConnection = true;
    }
    if (endConnection || !client.Connected)
    {
        stream.Close();
        client.Close();
        return;
    }
    Task.Delay(15).Wait();
}
```

Code Snippet 34: Second part of the `HandleConnection()` method

4.3.5 Node

Previously, we have described the inner workings of the Main Nodes, which contains all the logic of the DCR graphs engine, propagation, and so on. We do, however, also have the Node program which contains code pertaining to sending requests to their related Main Node. The code for this program is much less complex and will therefore only be described briefly.
The Node program consists of 4 classes. First is *Client*, which is almost identical to the one described in paragraph 4.3.4.1, except it is not asynchronous and therefore does not require a semaphore as property. It also does not keep the connection open after a reply has been received, as many messages in succession are not expected to be as common from the Node.
Secondly, the *Config* class is a miniature version of the one found in subsubsection 4.3.1. The Node config file only contains "[Properties]" and thus, these are the only properties that are parsed. The config also contains a single `Client` instance, representing the Node's local Main Node. Third is the *Node* class, which contains methods to parse input in the terminal, and to create request messages based on this input. The class also has a property `events`, which is a string array used to contain the names of which events the Node is allowed to execute. These are direct references to *EventInternal* names located on the Main Node, and are received by calling the `GetEvents()` method in this class. Finally, the *Program* class simply holds the `Main()` method which creates an instantiation of *Config* and *Node* and runs an infinite loop of reading user input and handling user input with the `Node.HandleInput()` method.

5 System Quality Attributes

5.1 Availability

System failures were a significant concern when designing the distributed system, because while some parts of the system, i.e. a Main Node, may execute events independently if they do not have relations to each others events, a single Main Node's unavailability may on the other hand, spread to other Main Nodes who require some information from that Main Node. This is especially true for propagating requests such as a request to find out whether or not a DCR graph is in an Accepting state. If the only Main Node knows three other Main Nodes becomes unavailable, those three additional Main Nodes will in turn, become unavailable.
To improve the availability of our system, we have done a few things. First, as has briefly been mentioned in subsection 4.3, we use semaphores in several places. One such place was when using the client to open TCP connections and sending messages to other Main Nodes, where we ensure that if we are currently using that client instance to send and receive something, any other pending send requests will only be done after the current one is complete. This is required, as reading the reply is not thread safe, and is done by locking a semaphore whenever we enter the `Client.SendAsync()` method, and unlocking whenever the method exits. Thus, the data that the client receives back as reply for its request is always the correct reply, as no other requests may be sent asynchronously, to the same Main Node.

5.2 Modifiability

The system is separated into two different layers, a network layer consisting of the *Client* class and the *Listener* class and a business logic layer which contains the logic of DCR graphs. The two layers are coupled together via abstraction using interfaces, where they do not depend on the direct implementation details of the class and its methods. This increases the modifiability of the system where the interfaces appear, since components which implement the interfaces can easily be interchanged. There are four interfaces *IGraph*, *ILog*, *IClient* and *IConfig*. *IConfig* is the first module to be created and is passed on to the rest. This is done via constructor injection, where an *IConfig* is provided in the constructor of a class which depend on it. This way we do not have to instantiate a *IConfig* class inside the dependent module and only one instance of the object is needed.
This results in loose-coupling between the modules which implements the interfaces, as they depend on the abstractions instead of the details. There exists tight-coupling in the system as well, particularly between the *Parser* and *EventForeign* classes and between *Graph* and the *EventInternal* class. The tight-coupling in these places are a result of not using abstractions and the instantiating of the objects directly in the dependent class.

With the *Config* class and the `Graph.Load()` method, a DCR graph can be easily loaded by the program and all the necessary events, relations and Main Nodes will be created automatically. The

Config class and the `Graph.Load()` method read from specified files, and this enables a quicker setup of the system by allowing file paths to be changed. The user will still have to recompile the program every time the user wishes to change which file is loaded. We argue that this is still quicker than having to manually create the graph in the `Program class`. The graph XML file can be generated using an online system (dcrgraphs.net), but the config file is manually edited by the user. The ability to specify the location of the file enables the user to have multiple files ready and the user can change the files by changing the path to the file and recompiling the program. The specifics of this is covered in section 7.

5.3 Performance

We have generally prioritised performance over disk space used, which can be seen when looking at our implementation that often takes advantage of the C# Dictionary class. We chose to do this because we judged that the amount of disk space required to contain a local DCR graph would be very low compared to the amount of messaging required. Thus, we make messaging and processing faster by indexing Dictionaries in constant time, rather than having to iterate through e.g. an array in potentially $O(n)$ time.
We also drastically increase performance by the fact that each Node has its own thread on the Main Node, meaning Main Nodes may handle several requests in parallel, as long as they do not affect each other, or the same events - i.e. in execution requests. Sending messages to other Main Nodes is also done asynchronously on these aforementioned threads, such that we can send a request to many different Main Nodes at the same time, and then wait for them all to reply back afterwards. We opted for asynchronicity instead of creating a new thread for each request, as the overhead required to make a new thread is not worth it, considering the `Client.SendAsync()` method is not very computationally expensive.
Another performance optimisation is the way we limit the amount of messages sent to other Main Nodes. As seen in subsection 4.3, we combine different marking update requests to the same Main Node using `Graph.GetPeerMessages()`, which limits the amount of messages we have to send for a given request to a given peer to always be only one message. For requests such as retrieving a global log of executions or whether the graph is accepting, we can also limit the messages required by a single Main Node to send, by limiting the amount of Main Nodes it knows. This is done innately by the design, which does not require a Main Node m_1 to keep track of any external markings, meaning certain tasks must be delegated to other Main Nodes. This design does result in an impact in performance as messages in this case have to arrive at the external Main Node m_2 with knowledge of these aforementioned markings, and m_2 then must send an additional message to a third Main Node m_3 - as opposed to m_1 simply messaging both of those Main Nodes asynchronously. This does however spread the workload of handling messages and opening sockets across several Main Nodes in the system, and thus we do not have to have any additional updates or messages to and from a Main Node in order to determine other Main Nodes' markings and relations, improving overall performance.

As has already been mentioned, duplicate propagation requests may be received by a Main Node in some cases. We reduce messaging by preventing a duplicate request from being performed, by saving an ID of a given propagation request, and upon receiving one we compare it to the received ones. We also highly reduce the amount of duplicate requests to be made in the first place, by including a list of peer names that have already received the request, in each of our propagation request messages, letting the recipient know who not to forward the propagation to.
When a Main Node receives a "REVERT", we also know by the communication protocol's design, that it will inevitably be unblocked. Thus, we do not need to send the "UNBLOCK" message, as Main Nodes will automatically unblock themselves after they have reverted their markings, and the markings of the Main Nodes which they propagated the previous "BLOCK" and "EXECUTE" requests to. In addition to this, as mentioned in subsection 5.1, in our Main Node we chose not to always close a socket immediately after sending a message. We decided this, because we often send several messages in a row - such as in the event of an execution where we first must send a BLOCK, then EXECUTE, and then finally UNBLOCK. Closing the socket would require us to dispose of the socket, reinstantiate a new identical socket, and reconnecting - whereas simply letting the connection stay open allows us to instantly use it to allow almost instantaneous messaging. Instead, as mentioned in paragraph 4.3.4.1, we close sockets after receiving or sending a termination character. Another performance optimization we have done is that we always perform the checks that are fastest, and use short-circuiting where it is logically possible, in order to potentially prevent doing any long checks. An example of this is when we are trying to execute an event, we initially see whether the event we want to execute is enabled with `Graph.Enabled()`, and if it is not, then we do not try to find out whether we are able to block any additional events with `Graph.TryBlockInternal()`.

5.4 Security

As the system has been made using C#, a high level programming language, we trade performance for security in that we perform run-time checks to ensure we, for example, do not read memory that we should not, by reading past the length of arrays. The language also does not expose any unsafe methods or allow direct manipulation of pointers, unless we intentionally use the keyword *unsafe*, which we do not do in our program.

We also use Transmission Control Protocol (TCP), which by itself helps guard against many network-layer attacks. Furthermore, the design of our system, where a Main Node is located on a company server, allows the given company to adjust their security policies to fit their needs. By doing this, we partly leave security to the company.
Other than this, we did not focus much on security although security is beneficial, it is not our speciality nor the focus of our thesis.

5.5 Testability

As mentioned in subsection 5.2, the four interfaces help in making the system more testable. The increase in testability comes from modules being able to be isolated and this enables the use of unit testing. The way we isolate the modules is with a technique called mocking, where we can use a framework called `moq` to automatically produce implementations of the interface and in doing so, we can isolate each module from the others. More on the specifics of mocking and testing strategy in section 6.

When unit testing, it is desirable to isolate modules so that the test is only testing the functionality of a given method and not the dependencies of the method. A tightly-coupled system will have difficulty in doing so, since a method or class will depend directly on the implementation details of its dependencies. With the use of interfaces we use abstraction to hide the details and we are thus able to inject any implementation of the interface into the test using mocking.
As noted in subsection 5.2 there are still parts of the system which are tightly-coupled and in these cases we can not abstract the details away. Thus, we still depend on the implementation of the dependencies and as such unit testing becomes more challenging. Since there are no interfaces to mock we must use the dependencies. A good unit test tests the functionality of a method or class and not the dependencies, and the tightly-coupled code makes this difficult to do. This also makes the tests fragile - if the dependencies gets modified, the test might not work and will have to be re-written. It also makes it difficult to test the isolated behaviour of a class or method, since they will be affected by how its dependencies are implemented and if these change - the behaviour can change. This makes the implementation rigid and hard to modify without breaking classes that depend on it.

Testing the distributed part of the system was more difficult, since testing usually occurs on a single machine. One way to test the distributed features requires at least 2 machines to run the system on - so that we would have two Main Nodes, one on each machine. This is difficult to do automatically, but is possible and has been done manually. It was possible, however, to test the entire system automatically by having several Main Nodes and Nodes on the same machine and having them listen and send to different ports on the localhost. We can then make Nodes send requests to Main Nodes, which then propagates the message throughout the system and finally returns it to the requester Node. System correctness is then verified by looking at the data of the system, as well as the messages that the Node receives. With this approach, however, it is not possible to cover edge cases - this is elaborated upon in section 6.

5.6 Usability

All user input is given via the command-line interface and users interact with a Main Node through a Node. When the user inputs a request in the Node, the Node will handle the request by sending a message to the corresponding Main Node. If the Node receives an unknown command, it will report it to the user in the standard output. Usability in a command line window can be limited, since the only way to communicate an error to the user is via text or unhandled exceptions, which will crash the system. From a usability standpoint - crashing the system is not a great option and therefore we must rely on communicating through text in the command line window.

When a user interacts with the Node solution through the command line window they can enter a variety of commands, as outlined in section 4, and the system will respond to these. When an unknown command is entered, the system informs the user of a help command, which lists the available commands to the user. This is implemented so that users will have an easier time when they first work with the system. When a correct command is entered in the command line and the request has been handled by the Main Node, the result of the request will be displayed to the user, informing them of the result of the request. Whenever a request is unsuccessful, the system will display the same failure message no matter what went wrong with the request. This can hurt usability, since users do not know what went wrong but simply that something went wrong. A more constructive way of reporting the error would be to tell the user what went wrong with the request, so that the user knows if it made a fault or if it was a fault in the system.

We chose to use strings as the input method for users, since it will be easier for user to understand which command they are using.
The bar of entry into the distributed system can also be quite high and will most likely require an IT professional to set it up. First, one must have a computer which can run the Main Node part of the system. This can either be a dedicated computer or it can run besides the Node program on a user's machine. Second, the computer for the Main Node must have port forwarding enabled. Third, other peers must know your Main Node IP and which port it is listening to, by defining them in the .ini file, along with the other settings (see section 7). These three requirements sets a rather high bar in order for users to participate, since they must inform others that they are participating and where they are participating from.

Since we cannot interact directly with a Main Node while it is running - usability in the Main Node is concerned with how easily a user can set up a DCR graph prior to the system starting. Before the Main Node is started, the user must create the DCR graph, the events in the graph and the relations between events. This is quite easily done using the dcrgraphs.net system, although the naming of events (activities) must be of a special format, as outlined in subsubsection 7.1.2. After exporting the graph as .xml, the program automatically converts the global graph into a local one - which we argue highly increase the usability of the system.

6 Testing

For testing our distributed DCR graph engine, we decided to perform three kinds of tests; unit-, integration- and system-testing. In doing so we've followed a few practices, which will be outlined here along with the different frameworks used for testing the code.

Testing software revolves around asserting that the software under test meets the expectation of the developer and meets the requirements. Software is generally said to fulfil its requirements if all its tests pass and thus if one fails the software is said to not meet the requirements. In writing our tests, we followed a test pattern known as `AAA`[3], standing for Arrange, Act and Assert, as seen in [9]. Meaning you arrange the preconditions of the test, perform the testable act and assert the expectations. In the "arranging preconditions" step, everything needed to perform the act is set up. This includes data structures, mocks and objects. The method under test is executed, and the result is usually saved for the next step. The last step asserts that the result meets the expectations and if there are any side effects of the method execution, that they meet the expectations as well.

In addition to the test pattern described above, the naming of the tests follow a specific pattern as well. This pattern is as follow: `Method_Case_Expectation`. Where `Method` is the method under test. `Case` is the case, that the method is being tested under. An example of this is if the method under test takes a string array as input. If the array is empty, then the `Case` part of the name would be that the input array is empty, informing the reader what is special about the test. `Expectation` is the expectation for the result. To continue the previous example, the method under test could return false when given an empty array, then the expectation would be false and the name of the test would be `MethodUnderTest_EmptyArray_False`.

In order to isolate methods and classes during testing, we have used a technique called mocking[4]. Mocking or mocks are a test double for dependencies and are a fake implementation of the dependencies' interface. In this fake implementation, the programmer can set up the different methods of the dependency and define what should return given any or a specific input. This allows for an isolated controlled environment where the programmer has complete control of the dependencies for the method under test. In addition, mocks allow the assertion of method calls, which means that if the method under test calls another mocked method, the mocking framework can verify that the mocked method was called as expected. We cannot mock fields, and thus some of our unit tests have dependencies on *EventInternal*, *EventExternal*, and *EventForeign* - these events are however very simple data structures with few to no local methods. In our testing we use the `moq`[5] framework for C#. Mocks are mostly used in our unit tests, since it is desirable to have the method completely isolated. They also occur in the integration testing where we mock modules which are not part of

[3][9], p 148
[4][9], p. 160
[5]Moq Github

the integration test. For example if we have an integration test between *Config* and *Graph*, then the log for the *Graph* class will be mocked. In unit testing, the mocks are set up using a setup method from the `NUnit`[6] testing framework. This method is run before every test and creates a new mock. Not all the tests require a setup method since no overall mocking or instantiating of dependencies is required. If required, additional mocking is done doing an `Arrange` part of the test to arrange preconditions for the test. Mocking via interfaces also prevent brittle tests, meaning that the tests wont have to change if the concrete implementation changes, since they depend on the abstractions instead of the details.

The *Client* and *Listener* classes were only scarcely tested because of the difficulty of mocking the *TcpClient* class of the .NET socket library. The difficulty comes from the *TcpClient* not having an interface and the inability to mock implementations. We instead rely on the tests for the parser and message construction to verify that the connection work. We do this by verifying that a message is constructed and parsed as expected in their respective classes. We conclude, that if a message is constructed as expected and sent to the *Clint*'s `SendAsync()` method correctly, then it will also send the message if a connection is established to the Node or Main node. When that message is sent to a Main Node we conclude that if the *Parser* parses the message correctly, then it is working as expected.

A folder called `TestAssets` is needed for the tests to function properly. This folder contains all the necessary XML test graphs and test .ini files in order for *Config*, *Graph*, integration- and system tests to run. This is included with the system and should be located in the $DDCR \longrightarrow bin \longrightarrow debug \longrightarrow netcoreapp3.1$ folder. In other words, it should be located with the executable for the test project.

6.1 Unit Testing

We used a white-box testing technique for our unit tests, where all details surrounding the code is known and the tester have full access to the code under test. Our aim was to test each branch at least once and if `if-statements` had multiple conditionals, then each combination of the conditionals would be tested. Meaning if an `if-statement` has two conditionals, then four different tests are needed for it. In addition to this, all classes containing business logic should be tested along with all the methods in those classes. As previously stated, this means that the *Client* and *Listener* is only tested sparingly.
The following are two examples of different unit tests showing how `Moq` is used and how the tests are arranged.

```
[Test]
public void TryExecuteExternal_SuccessfulExecuteManyPeers_True()
{
    // Arrange
```

[6]NUnit.org

```
    Guid id = Guid.NewGuid();
    Dictionary<string, Task<string>> tasks = new Dictionary<string, Task<string>>();
    Mock<IClient> mockClient = new Mock<IClient>();
    mockClient.Setup(x => x.SendAsync(It.IsAny<string>()))
            .ReturnsAsync("SUCCESS").Verifiable();
    for (int i = 0; i <= 4; i++)
    {
        config.MainNodes.Add("mainNode" + i, mockClient.Object);
        tasks.Add("mainNode" + i, Task.FromResult(""));
    }
    // Act
    var result = graph.TryExecuteExternal(tasks, id);

    // Assert
    var expected = true;
    mockClient.VerifyAll();
    Assert.AreEqual(expected, result);
}
```

Code Snippet 35: Execute test example using Moq and NUnit

In **Code Snippet** 35 we see at unit test for the `TryExecuteExternal()` method. The case is that it successfully executes with more than one Main Node and the expected outcome is true. The test starts out by arranging the `Guid` and a Dictionary tasks, used as input to the method under test. It then creates a mock of the *IClient* interface and sets up the `SendAsync()` method, so that when `SendAsync()` receives any string it will return "SUCCESS". It then proceeds to add five Main Nodes in a for-loop to the `config.MainNodes` dictionary, where the value of each Main Node is the mocked client returning "SUCCESS". After this the arranging of preconditions is done, we proceed to the Act phase, where the `graph.TryExecuteExternal` is executed and the result is saved in a variable. Next is the assert phase, where we assert that the method in the mocked *IClient* was called the expected number of times and that the expected result is equal to the actual result.

```
[Test]
public void TryBlockInternal_ResponseSecondEventExcluded_True()
{
    // Arrange
    graph.AddRelationInternal(eventFrom.Name, eventTo.Name, Relation.Response);
    graph.AddRelationInternal(eventTo.Name, thirdEvent.Name, Relation.Milestone);
    eventTo.Included = false;

    //Act
    var result = graph.TryBlockInternal(eventFrom, id);

    // Assert
    var expected = true;
    var expectedSecondEvent = new Tuple<bool, Guid>(true, id);
    var expectedThirdEvent = new Tuple<bool, Guid>(false, Guid.Empty);
    Assert.AreEqual(expected, result);
    Assert.AreEqual(expectedSecondEvent, eventTo.Block);
    Assert.AreEqual(expectedThirdEvent, thirdEvent.Block);

    Assert.AreEqual(true, graph.Blocked[id].Contains(eventTo));
    Assert.AreEqual(false, graph.Blocked[id].Contains(thirdEvent));
}
```

Code Snippet 36: TryBlockInternal test example using Moq and NUnit

In **Code Snippet** 36 we see a test targeting a specific branch in `TryBlockInternal()` where we have three events and the third event is not blocked if the second is `excluded`. In the arrange step we set the relation between the three events. These events are created in a `Setup()` method, as they are used for nearly every test in the test class. It also sets the second events `Included` marking to false, since it is needed to hit the branch in the code under test. In the act step, the method is executed and lastly, in the arrange step, the different assertions are done. These include whether the result of the method is as expected - in which case the method returns true. It also asserts that the events are blocked correctly and that the second event is contained in the `Graph.Blocked` dictionary and that the third event is not.

6.2 Integration Testing

Our integration testing strategy was a bottom-up strategy, where low level classes are tested together first and higher level classes are tested last. One of the advantages of this approach is that it is easier to locate errors early in the testing since the lowest level classes are the ones which are tested first. The downside, however, is that the larger control classes are tested last. For our system, this means that *Parser* and *Graph* is the last to be introduced in the integration tests and their errors will therefore be exposed last.

As with the unit tests, the aim of our integration tests is to hit every branch of a method under test. Most of the integration tests are similar to the unit tests but with mocking removed, which means that their dependencies are introduced in the test. This way, we verify that the method under test behaves as expected with its dependencies. The naming of the integration test classes are chosen for which modules are tested together, an example is `ParserGraphTests` where the *Parser* and *Graph* is tested together.

The following are two examples of integration tests:

```
[Test]
public void ParseMainNode_BLOCKCaseThreeValidEventsCannotBlockInternal_UNAVAILABLE()
{
    //Arrange
    graph.CreateEvent("e1", "l1", "Node1".Split(), true, false);
    graph.CreateEvent("e2", "l2", "Node1".Split(), true, false);
    graph.CreateEvent("e3", "l3", "Node1".Split(), true, false);
    graph.Events["e1"].Block = new Tuple<bool, Guid>(true, Guid.NewGuid());
    graph.Events["e2"].Block = new Tuple<bool, Guid>(true, Guid.NewGuid());

    Parser parser = new Parser(graph, config);
    Guid id = Guid.NewGuid();
    DateTime date = DateTime.Now;

    string eventToBlock = "e1 include\n"
                        + "e2 exclude\n"
                        + "e3 condition-";
    //Act
    string result = parser.ParseMainNode("BLOCK " + id.ToString() +
            " " + date + "\n" + eventToBlock);

    //Assert
    Assert.AreEqual("UNAVAILABLE", result);
}
```

Code Snippet 37: ParseMainNode Integration test example

In **Code Snippet** 37 we see an integration test of *Graph* and *Parser*. The method under test is `ParseMainNode()`'s BLOCK case, the case is that there are three events and two of them are blocked when the method tries to block them. The expected result is "UNAVAILABLE".
The graph object is setup in the `Setup()` method. The test starts out by arranging three events using the `graph.CreateEvent()` method and then proceeds to manually block the first and second event. It then finished the arrange step by creating the *Parser* object and the id, time and input string used for the method call. In the act step it executes the method under test and saves the result in a variable. Lastly, in the assert step it compared the expected result with the actual result.

```
[Test]
public void Execute_SuccessfulWithExternalEventAndInternalEvent_True()
{
    // Arrange
    var mockClient = new Mock<IClient>();
    config.MainNodes.Add("peer", mockClient.Object);
    ev.IncludesExternal.Add(eventExternal);
    graph.AddRelationInternal(ev.Name, evSecond.Name, Relation.Response);
    mockClient.Setup(x => x.SendAsync(It.IsAny<string>())).Returns(
            Task.FromResult("SUCCESS")).Verifiable();

    // Act
    var result = graph.Execute(ev.Name);

    // Assert
    var expected = true;
    var executed = true;
    mockClient.Verify(x => x.SendAsync(It.IsAny<string>()), Times.Exactly(3));
    Assert.AreEqual(executed, ev.Executed);
    Assert.AreEqual(true, evSecond.Pending);
    Assert.AreEqual(new Tuple<bool, Guid>(false, Guid.Empty), evSecond.Block);
    Assert.AreEqual(expected, result);
}
```

Code Snippet 38: Execute Integration test example with a mock of *IClient*

In **Code Snippet** 38 we see an integration test for the `Execute()` method in *Graph*. It uses a Mock for the external events, so that the return value can be controlled. The case for the test is that the events passed to the `Execute()` method has both internal and external events as relations and that blocking and executing these are successful. The expected result of the test is that `Execute()` returns true. In the arrange step of the test, an external event is added to the `IncludesExternal` HashSet for the executing event. It then adds a internal relation between the executing event and an internal event. Lastly in the arrange step, it creates a mock of the *IClient* interface, adds a Main Node to `config.MainNodes` and sets up the mocked *IClient*'s `SendAsync()` to return "SUCCESS" on any input string. In the act step, the method is executed and the result is saved in a variable. In the assert step we verify that the the mocked *IClient*'s `SendAsync()` was called exactly three times. We also assert that the executed event's executed marking was set to true and that the pending marking of the affected event was set to true, since the relation between them was a response relation. Lastly we assert that the second event is not blocked anymore and that `Execute()` returned true.

6.3 System Testing

Finally, we have both manually conducted- as well as constructed manual and automatic - tests of the entire system. The manual testing was performed using a combination of Main Nodes and Nodes located both locally to one another, as well as externally (on a seperate computer, on a seperate IP Address). All the automatic tests, however, are all done locally. We have manually tested a variety of graphs, attempting to cover all edge cases and scenarios. An example of this, is manually blocking requests at certain points in a message sequence, and seeing whether our error handling and recovery measures are working as intended. Our automatic tests are not able to test the system in this matter, in between message sequences, as the connection to a localhost obviously cannot be severed. Moreover, the NUnit testing framework does not allow us to instantly, forcefully destroy threads. Instead, our automatic tests test the system when all Main Nodes are alive, as well as when one Main Node stops listening for messages (i.e. dies). Note that due to the nature of these tests, they do not properly follow the AAA pattern. The graphs used for the system tests are :

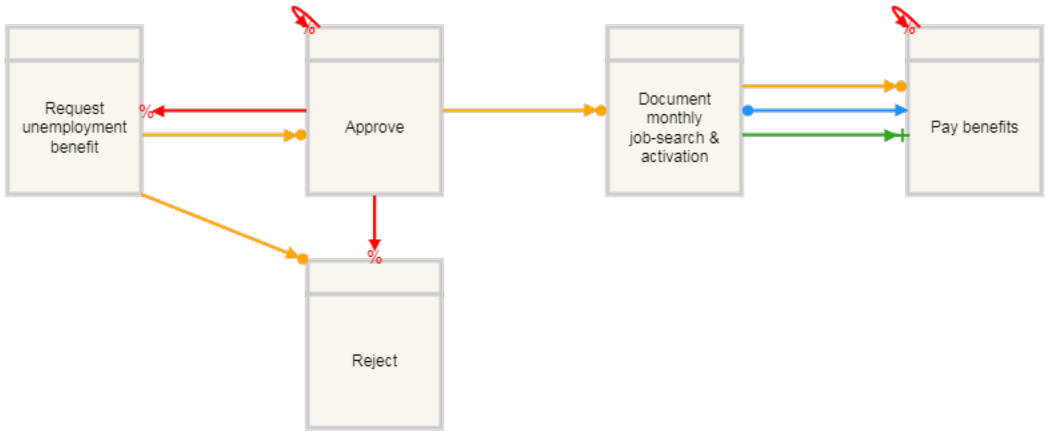

Request unemployment benefit
Approve
Document monthly job-search & activation
Pay benefits
Reject
%

Figure 12: Graph for the first System Test, from [11]

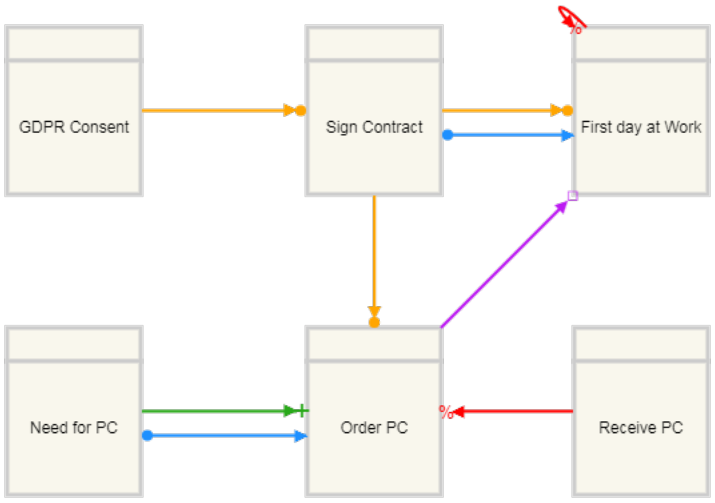


Figure 13: Graph for the second system test, from [12]

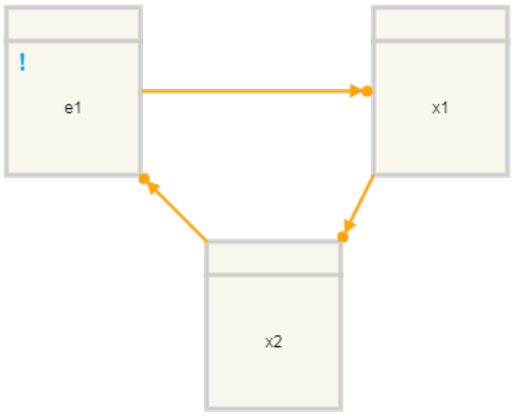


Figure 14: Graph for the third system test, common deadlock DCR graph.

6.4 Methodology of the System tests

In Figure 12 we see the graph used for the first system test. This graph is split between two Main Nodes, where the first Main Node has the `Request unemployment benefit`, `Approve` and `Reject` events and the second Main Node has the `Document monthly job-search & activation` and `Pay benefit` events. The graph is taken from [11]. At the start of the test, everything is set up for the two Main Nodes and their respective Nodes, along with the config files and the graph file being loaded. Before any event is executed, all relevant markings are asserted to check if the graph is created as expected. After the markings have been asserted, the test proceeds to execute the first event - `Request unemployment benefit` and afterwards asserts any relevant marking changes. It then attempts to execute the `Pay benefit` event, which returns false, because of the condition relation targeting it from a non-executed `Document monthly job-search & activation`. Afterwards, `Approve` is executed in the first Main Node before asserting that `Approve`, `Reject` and `Request unemployment benefit` are all excluded by the execution of `Approve`. `Document monthly job-search & activation` is then executed, since the condition from `Approve` is no longer active and after the execution we assert that `Pay benefits` are included, pending and has a condition equal to zero. Last of the events to be executed in the test is `Pay benefits`, which this time return true. We then assert that it is no longer pending and that it excluded itself. Lastly, we query for the accepting state, which is expected to be true due to no pending events, and we query for a log and assert that the expected execution order, which is the one detailed above, is accurately represented in the log.

In Figure 13 we see the graph used for the second system test. The graph is split between three Main Nodes, each with one Node with permission to all events associated with that Main Node. The first Main Node is the largest and has 3 events, namely `GDPR consent`, `Sign Contract` and `First day at work`. The second Main Node has one event - `Need for PC`, and the third Main Node has two events, `Order PC` and `Receive PC`. The graph is taken from [12]. As with the first test, it starts off by setting up the Main Nodes and Nodes with config files and the graph XML file. It then asserts that all events belong to the expected Nodes and that the markings are set correctly. The test then proceeds to execute `GDPR consent` and asserting that the expected markings change following the execution. It then attempts to execute `First day at work`, but fails due to the condition relation from a non-executed `Sign Contract`. The test then executes `Sign Contract` and asserts its expected marking changes. At this point it will attempt to execute `Need for PC`, which is the most interesting case in this graph, due to the milestone relation from `Order PC` to `First day at work`. This milestone relation is activated when the `Need for PC` is executed because of the response relation between `Need for PC` and `Order PC`, and since all three events are on different Main Nodes the system will have to propagate the BLOCK message from the receiving Main Node to the next Main Node down the line. Meaning that Main Node two, which is hosting `Need for PC` sends a BLOCK to the third Main Node, which is hosting `Order PC`. The third Main Node must then propagate to the first Main Node, which is hosting `First day at work` in order to block that event, due to the impending activation of the milestone relation. When the block is secure on `First day at work` and `Order PC`, the execution can continue. The test proceeds to execute the rest of the events before querying for the log and the accepting state and asserting against expectations.

7 How to run

The following section will explain how to set up the system from scratch. We have also included a Manual Testing Suite, with already set up initialisation files and graph. For more information on how to use this testing suite, refer to the readme.txt inside of it, as well as the this section.

7.1 Main Node Setup

In order to set up a Main Node and its associated local graph, two files are needed: A MainNode.ini and graph.xml. The .ini should be located in the same folder as the program executable, and the graph should be located in one directory below the executable. This is easily modified, as mentioned in subsection 7.3.

7.1.1 MainNode.ini

As mentioned in subsubsection 4.3.1, the config is separated into several categories. If a setting is written under a category they do not belong to, an error will be thrown. Each category will be detailed below. Example values are given in quotation marks, but note that the quotation marks should not be included in the config.

7.1.1.1 [Properties]

The format for properties is: property=value
Note, that there is no space in between the equals symbol, property and value. If a white-space is included in a string, then the white-space will be parsed as part of the string.

- Name
  The name of the Main Node. This should match the name in the graph, which will be elaborated in subsubsection 7.1.2. Names may contain letters and numbers.
  Example values: "Bob" or "MainNode1".

- ListenPort
  The port which the Main Node should use to receive incoming requests˙
  Example value: "5005".

- ClientTimeoutMs
  Main Nodes keep their connections with other Main Nodes alive. This setting defines the amount of time it will be kept alive, in milliseconds. It also defines the amount of time to wait for replies to TCP requests before timing out.
  Example value: "5000".

- MaxConnectionAttempts
  If another Main Node stops responding to messages after it has been blocked, this Main Node will attempt to unblock it. This setting defines how many times it will attempt doing this before giving up. Example value: "5"

- Culture
  This setting controls how dates should be displayed, e.g. when getting a log of all elements. This setting should be shared across all Main Nodes. Example values are: "en-GB" and "en-US".

7.1.1.2 [MainNodes]

This category should contain the names, IP addresses, and ports of the other Main Nodes. Note that all other Main Nodes in the entire network are not required to be listed here; only the ones that the Main Node is expected to send to. If it is uncertain which Main Nodes are required, you can list all of them.
The name of the Main Node, as well as the port should match the `Name` and `ListenPort` given in its own `MainNode.ini` file, under **[Properties]**.
Format: Name=IP:Port
Example value:

```
"MainNode2=127.0.0.1:5006
 MainNode3=127.0.0.2:5007"
```

7.1.1.3 [Nodes]

This category should contain the names and IP addresses of its own Nodes, whom are expected to request execution of events.
Format: Name=IP
Example value:

```
"Node1=127.0.0.3
 Node2=127.0.0.4"
```

Note, if a node is run on the same computer as the Main Node then use the localhost ip: 127.0.0.x. If, however it is run on another computer but in the same local network - use the internal ip: 192.168.0.x. The internal ip can be found in the command terminal by using the `ipconfig` command on windows and looking for IPv4 under "Local Area Connection".

7.1.1.4 [NodeEventMapping]

After the Nodes have been defined, this configuration specifies which local events a Node may request to execute. Several Nodes may have permission to execute the same events. Events are separated by spaces, as seen in the formatting:
Format: Name=Event1 Event2 Event3
Example value:

```
"Node1=EventOne EventTwo
 Node2=EventOne EventTwo EventThree"
```

7.1.2 graph.xml

The graph should be a global graph downloaded from dcrgraphs.net, using the "export as XML" functionality. The Id's of each event (also called activity) in the dcrgraphs.net graph should follow a specific naming format: The name of the Main Node which should own the event, followed by an underscore, followed by the name of the event. **Event names should be unique**, and may contain letters and numbers.
Format: MainNodeName_EventName
Example value: "MainNode1_EventOne"

Note, that the id is specified under Options -> Advanced -> ID while editing an activity in dcrgraphs.net.

7.2 Node Setup

Nodes use a configuration file similar to the Main Nodes, albeit with much less options and only one category: [Properties]. Like before, this configuration file should be called Node.ini and be placed in the same folder as the Node program's executable.

7.2.1 Node.ini

7.2.1.1 [Properties]

- Name
  The name of the Node. This property is not currently used for anything in the program, as nodes are identified by the Main Nodes by their IP addresses, and is mostly intended to help the user see which Node a given config belongs to.
- MainNode
  This specifies the IP address and port of the Main Node that the Node should send requests to when interacting with the DCR graph.
  Format: Name=IP:Port
  Example value: "MainNode=127.0.0.1:5006"
- ClientTimeoutMs
  This specifies how long a Node should wait for replies to TCP requests, before timing out, in milliseconds. Same format as in 7.1.1.1.

7.3 Running and using the programs

The default file names and paths used for the .xml graph and .ini file may be changed by modifying the `configFile` and `graphFile` strings in *Program.cs*, and recompiling the program. The same goes for the Node and its .ini file.

Once all .ini and .xml files are set up appropriately, execute all Main Nodes and Nodes. Nodes may now send requests to their respective Main Nodes using the console. Available commands and their usage, also mentioned in subsubsection 4.3.5, may be displayed in the console by typing "HELP".

8 Conclusion

We have presented a design for distributing and executing a Dynamic Condition Response (DCR) graph throughout a peer-to-peer network consisting of different, asynchronously communicating actors. As briefly introduced in section 2, this design was built with a company setting in mind - such that the DCR graph represents a process shared between different actors of the network, where these actors refer to either different companies or different branches of the same company. Using a unique messaging design of locking or blocking all impacted events from being executed or modified prior to an execution, our design ensures that markings are always changed synchronously, in spite of the otherwise asynchronous system. The design allows executing events, getting the marking of local events, whether the global, distributed DCR graph is in an accepting state, and getting a global log of what date and time each event has been executed in the graph.
We have implemented this design in the object oriented programming language C#, using the .NET framework, and thoroughly tested its correctness through rigorous unit and integration testing.

8.1 Future work

Currently, Globally Unique Identifiers (GUID) are used throughout the distributed system in order to ensure safe execution of messages. There is, however, a very small chance that two different computers could generate the same GUID, which could potentially ruin the correctness of the system. An improvement we would like to make, is to make an extended GUID that is unique, e.g. by making the name of the Main Node that generated the GUID part of the GUID.
Another improvement to the system would be to make errors more visible, e.g. in form of more detailed replies on what went wrong instead of receiving an "UNAVAILABLE", which currently is the case regardless of whether no response was received to a sent message, or if the *Parser* failed to parse the message that was sent. We would also like to be able for our Node to get a local Log, rather than being forced to message all other Main Nodes in the system in order to get a global one.
As we have detailed in subsection 5.4, we put the majority of the responsibility on the actors in determining their security, and we otherwise assume trust between these actors. We recognise that the system might lack the security to be viable in a trustless scenario, which is something that could be improved.
Additionally, we would like to implement automatic, incremental saving of the local graph state after each state change, in order to recover from potential crashes. We would also like to allow for run-time graph modification by adding events and relations, which we expect to be achieveable by using the system's already existing semaphores and blocking mechanism.
While we have included the *Milestone* extension to DCR graphs from [1], we would like for the system to be able to support real world scenarios in better detail, using additional extensions, such as dynamically spawned sub-processes introduced in [3], as well as timed DCR graphs, which enable discrete time deadlines and delays, by extending the response and condition relations [10].

References

[1] "Distributed Workflow Execution on a Blockchain". Article, 8 pages. Mads Frederik Madsen, Mikkel Gaub, Tróndur Høgnason, Malthe Ettrup Kirkbro, Tijs Slaats, Søren Debois.

[2] "Safe Distribution of Declarative Processes". Article, 16 pages. Thomas Hildebrandt, Raghava Rao Mukkamala, Tijs Slaats. IT University of Copenhagen. 2011.

[3] "Hierarchical Declarative Modelling with Refinement and Sub-processes." Article, 17 pages. Søren Debois, Thomas Hildebrandt, Tijs Slaats. 2015

[4] "Distributed Dynamic Condition Response Structures." Article, 8 pages. Thomas Hildebrandt and Raghava Rao Mukkamala.

[5] "A Case for Declarative Process Modelling: Agile Development of a Grant Application System." Article, 9 pages. Søren Debois, Thomas Hildebrandt, Tijs Slaats, Morten Marquard.

[6] "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications." Poster 3 pages. Rüdiger Schollmeier. 2001.

[7] "What is UPnP & Why is it Dangerous?". Blog post. Andy Green. 3/29/2020. https://www.varonis.com/blog/what-is-upnp/

[8] "Peer-to-Peer Communication Across Network Address Translators". Article, 14 pages. Bryan Ford, Pyda Srisuresh, Dan Kegel.

[9] "Adaptive Code - Agile coding with design patterns and SOLID principles.", 2nd edition, isbn: 978-1-5093-0258-1. Book. 2017 Gary McLean Hall

[10] "Contracts for Cross-organizational Workflows as Timed Dynamic Condition Response Graphs." Article, 49 pages. Thomas Hildebrandt, Raghava Rao Mukkamala, Tijs Slaats, Francesco Zanitti. 2012.

[11] "Discovering Responsibilities with Dynamic Condition Response Graphs." Article, 15 pages. Viktorija Nekrasaite, Andrew Tristan Parli, Christoffer Olling Back, Tijs Slaats. 2019.

[12] "Digitalising the General Data Protection Regulation with Dynamic Condition Response Graphs." Article, 10 pages. Emil Heuck, Thomas Hildebrandt, Rasmus Kiærulff Lerche, Morten Marquard, Håkon Normann, Rasmus Iven Strømsted, Barbara Weber. 2017.