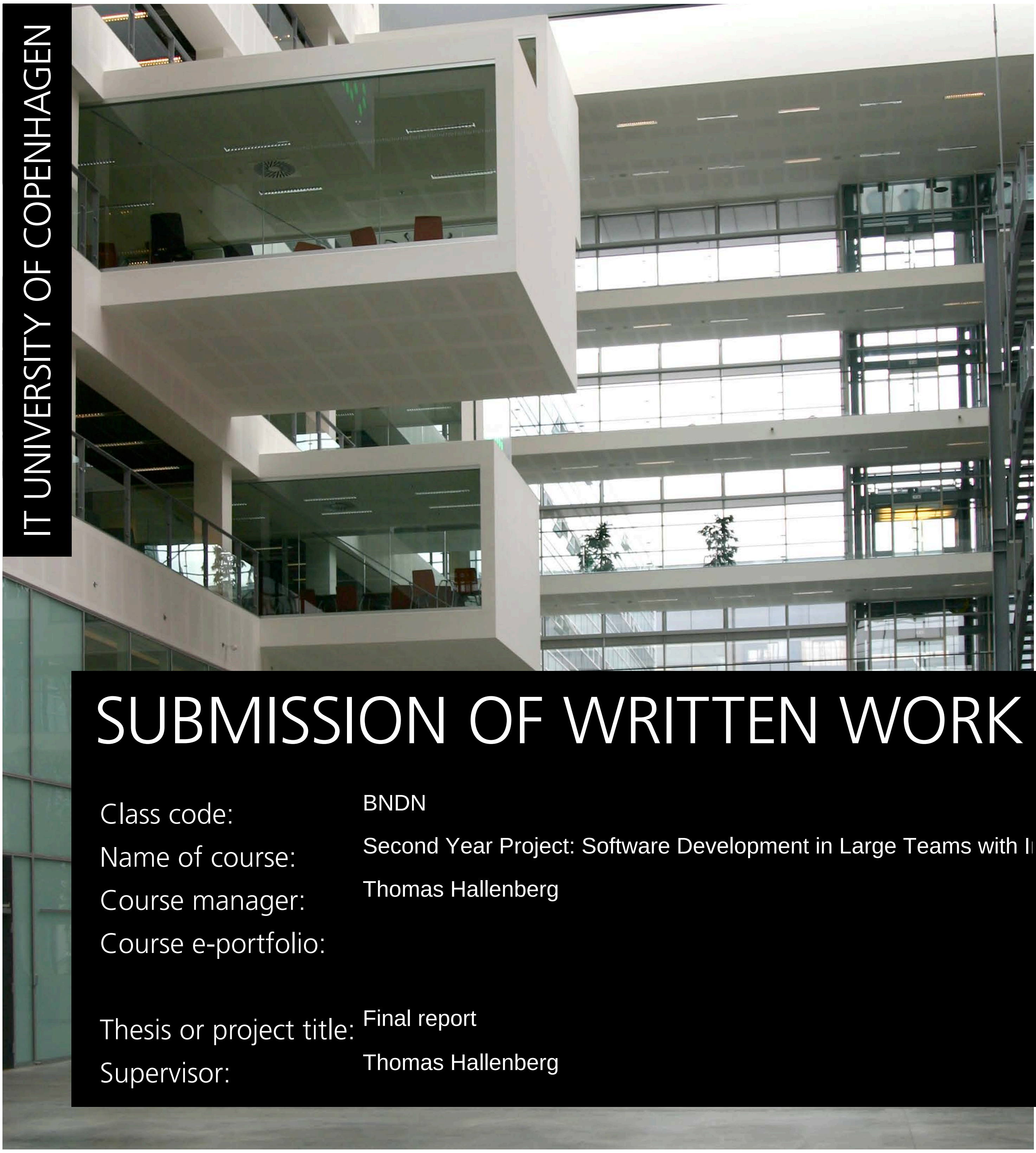


SUBMISSION OF WRITTEN WORK

Class code:	BNDN
Name of course:	Second Year Project: Software Development in Large Teams with I
Course manager:	Thomas Hallenberg
Course e-portfolio:	
Thesis or project title:	Final report
Supervisor:	Thomas Hallenberg

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Stinus Møhl Thomsen	10/07/1992	smot@itu.dk
2.	Patrick Evers Bjørkman	07/11/1992	pebj@itu.dk
3.	Aslak Vanggaard	23/05/1994	asva@itu.dk
4.	Christopher Blundell	10/08/1994	cnbl@itu.dk
5.	Jakob Lautrup Nysom	11/05/1994	jaln@itu.dk
6.			@itu.dk
7.			@itu.dk

IT University of Copenhagen

BNDN 2015

Second Year Project: Software Development in Large Teams with International Collaboration

BNDN Rapport

Aslak Vanggaard - asva@itu.dk
Christopher Blundell - cnbl@itu.dk
Jakob Nysom - jaln@itu.dk
Patrick Bjørkman - pebj@itu.dk
Stinus Thomsen - smot@itu.dk

May 15, 2015

Contents

CHAPTER 1

Background and requirements

1.1 Requirements Overview

The requirements have been given in two parts, the first one on March 3, 2015 and the second on April 21, 2015. The system for the second year project is a distributed peer-to-peer system that is capable of deploying a hosting generic DCR-graphs, and letting users access these through a REST API. There should also be a client program capable of communicating with the network, presenting these workflows to the user, and allowing the user to interact with them.

Functional Requirements

The system supports the following types of actors:

- user – The user should be able to execute executable events of a workflow that their role allows them to.
- The system should be able to model DCR-graphs, as presented in the requirement papers given.
- It should be possible to interact with a workflow in the peer network through a RESTful API.
- The system should be distributed and peer-to-peer, to the point where the events from a workflow, can be plased on different peers.
- It should be easy to set up a test example on a local host or a server.
- The system should implement some form of event log, eg: log when events are executed.
- The system should restrict the execution of events with some form of Role based Access Control, using the notion of ′roles′ from the DCR-graph specification.

- The system should employ some sort of concurrency control, related to its peer-to-peer nature.
- The system should include some degree of persistency, related to its peer-to-peer nature.
- The system should be able to handle generic workflows.
- The system should support a workflow shown in a provided draft paper *"Concurrency & Asynchrony in Declarative Workflows"*.

1.2 Workflows, DCR-Graphs and REST

Workflow

To explain what workflows are, we could imagine that we had a set of people with different roles. These people can execute activities as part of a process and these activities may only be executed upon satisfying a set of conditions. Such a condition could be that for one activity to be executed, another must be executed first.

DCR-Graphs

A method of representing a workflow is by the use of DCR-graphs. In DCR-graphs activities have three boolean values: Included, pended and executed. If an activity is included then it will become executable. If its pended it will work as a means of telling the users of the workflow that this activity is prioritized. Lastly if an activity is executed then its executed value will simply become true. The values become even more fundamental when it comes to the relations between activities. In DCR-graphs we have the following relations.

Relation types

- Condition
- Response
- Include
- Exclude
- Milestone

An activity can only be executed if it is included and all condition relations that points to it are executed or excluded. In the case that we have two activities known as A and B, if A is executed and has a response relation to B, B would become pended. Include and exclude would function in the same way except that instead of pended being set to true, include would be set to true or false. Being excluded simply means that the value of include should be false as include and exclude are just opposites.

In DCR-graphs the role of who can execute an activity is written in the top of the activity rectangle. The name of the activity is written in the middle and then the border around the rectangle will either be solid or dotted. Solid means its included while dotted means its excluded. An icon may also be within the rectangle. An exclamation mark means its pended while a check-icon means it has been executed.

REST

REST, *REpresentational State Transfer*[1] is the architecture we will be using for communication with our distributed workflow service system. The concept has a set of guidelines to follow in order for a system to be RESTful. One is that there should be a uniform interface which we satisfy by the use of HTTP Verbs such as GET (Receive), PUT (Update), POST (Create), DELETE (Removal) etc. The second requirement is that our system must be stateless. This means that the state must be contained within the request thus it could be that the state is within the URI or as part of the body. Using this architecture we can therefore preserve an uniform interface and stateless operation.

1.3 The hospital workflow

In order to create a DCR-graph based on the description we had received, we first read through the paper closely, marking down every word that seemed to relate to either an event, the actor starting it, or a relation between events.

After this we started a new graph on www.dcrgraphs.net and added the found events to it. The roles were also simply added this workflow. As for the relations, we decided pretty early on, that we would try to model a 'one-shot' workflow, meaning that we would disallow 'loops' in the workflow, and make it so that if an action on the graph would lead the user to a previously visited part in it, the user would have to start a new workflow instance. This was done as an attempt to decrease modeling complexity, as it would be much harder to ensure the correct state of the events in the workflow, if the user had the ability to visit these. We simply judged, that having to partially reset the workflow would increase the complexity too much, thus decreasing ease-of-use and navigatability. We also believed that it would be more useful to the 'hypothetical' user of our workflow, if they only had the choice between 'active' events (and not ones previously executed).

This 'one-shot' design of the workflow still lead to some complexity in the relations between events. Since this was before we realized the concept of 'logic gates'[2] can be used in DCR-graphs (which we used for the updated hospital graph), we instead used exclusion and inclusion to ensure that the correct events became available and unavailable. One of our problems was however, that this rampant exclusion of events also made it so that it became very hard to use conditions, since these relations are 'voided' once the event it leads from is excluded, and in the case of the hospital graph we had some events for which we needed an event 2 or more steps earlier to be a requirement for.

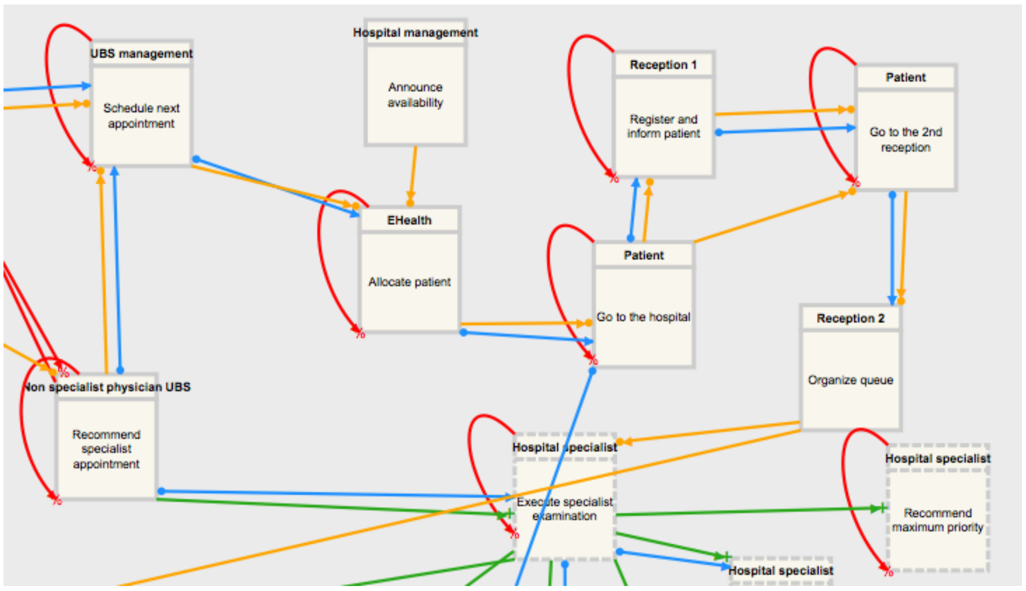


Figure 1.1 – A "condition" from "Recommend specialist appointment" to "Execute specialist examination". Since we are excluding "Recommend specialist appointment", the only way to make sure that "Execute specialist examination" only comes after that, was to make it so that "Recommend specialist appointment" has to include it instead

This had the unintended side-effect, that all the events that the 'pre-excluded' event was a condition for, now had to have a 'pre-exclude-then-include' relation since any conditions from an excluded event are ignored. This in turn gave us the problem that our chain of 'pending' events was broken, in a place where the user had to select one out of several events to execute in order to continue (a kind of logical OR relation). Since we did not use groups due to both not knowing very well about them, and not supporting them in our application, we could only put the event after these as pending, since otherwise a user of our workflow might think that all of these events had to be executed.

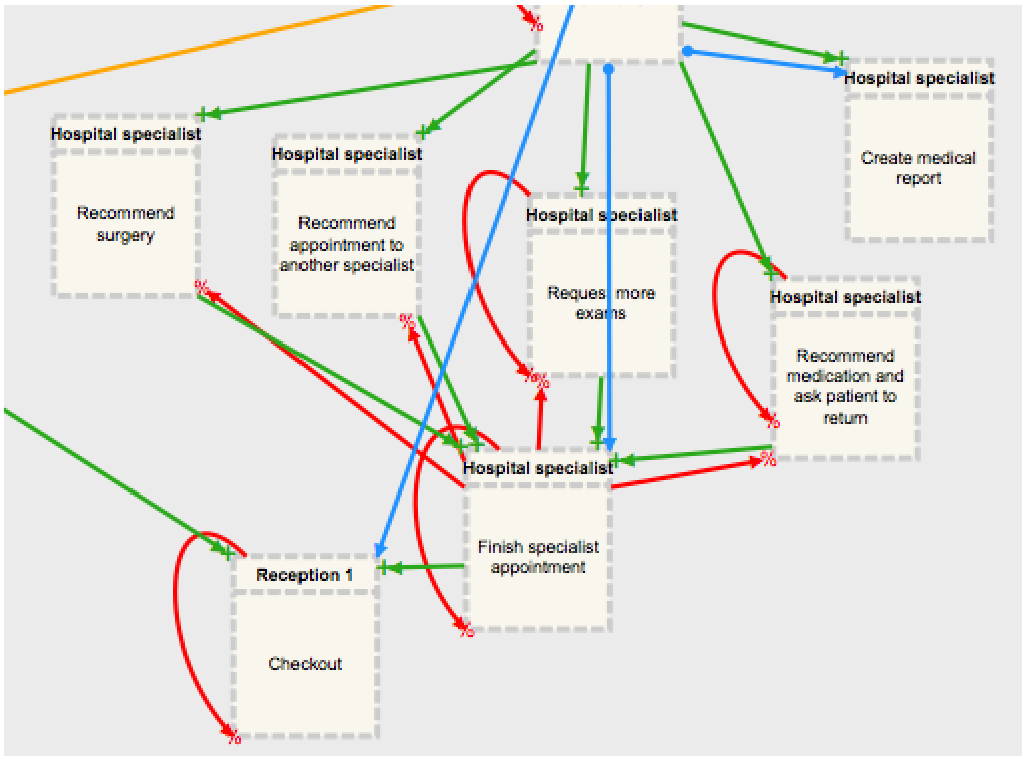


Figure 1.2 – The hospital specialist must "Recommend Surgery", "Recommend appointment to another specialist", "Request more exams" or "Recommend medication and ask the patient to return" in order to finish the examination, but not necessarily all of them. Since the events before and after them start out as excluded, "Finish specialist appointment" only becomes pending after the user have done one of them, so the user might incorrectly come to think that the workflow has been completed already at that point

In the end, our graph ended up correctly modeling the workflow as intended, with the exception of the broken 'pending' chain if the user took the "Recommend specialist appointment" branch of the workflow.

1.4 Initial plan and work division

The team approached this project in a very traditional way. The work method we chose was SCRUM, but it did not take effect until the second part of the project. Initially we sat down as a group and worked out a group contract to assure a mutual agreement on the quality of the product, expectations from one another and maintain* a high level of engagement throughout the group.

From here on we decided to continue with the first part without taking advantage of all the tools from SCRUM, but in a similar fashion. This meant we still had tasks and deadlines individually, but these were not documented with SCRUM. This workload was not quite equally divided, as some members took tasks upon themselves that were later discovered to take way longer than initially estimated, but were completed nonetheless.

During the first phase of the project, all of the groups members looked at the code. We were split into two groups of two, and a single man group. As soon as the first heap of code was produced and our program started to take form, people began to transcend* to tasks where they were the most effective, etc. certain parts of code requiring another level of skill were handled by the respective members. During this phase, meeting times and working days were almost always respected, with the occasional tardiness due to attention required by other courses and personal life.

During the second phase of the project, the methodology SCRUM was used entirely. Before commencing, a SCRUM meeting was held where all the initial details were discussed, SCRUM tools and documents were set up, and a general plan of the entire course was thought out. After this, the plan was to hold a retrospective SCRUM meeting at the end of each sprint, immediately followed by a SCRUM meeting commencing the second part. During these meetings, a SCRUM master was assigned on the day. It was required of each group member to list their available hours for the duration of the sprint, both so it was known when they would be working, and so the appropriate workload could be assigned to one another. After the initial SCRUM meeting, a list of the pending work tasks were listed that were expected to be completed during the sprint. The way the workload then was assigned, was that group members would pick out the various tasks they wished to work in accordance to their available listed hours. Again, the amount of time put in by each group member differed in accordance to their other courses and personal life. SCRUM greatly helped devise a plan and a general outline, and a clear division of work. However, some tasks were a bit more time consuming than others, resulting in the tasks being pushed into next weeks sprint. A detailed description of the work division can be found in the appendix.

CHAPTER 2

Elaboration of workflow requirements

2.1 Lifespan of our DCR Graph

Requirements for the hospital workflow

Eduardo Alves Portela Santos our international collaboration partner, gave us a paper described more in section 1.3. In this paper he gave the following requirements to our hospital workflow, as interpreted by us:

- The patient will first visit the UBS and the UBS can recommend further treatment or examines.
- The hospital will inform of available appointments and examines.
- When the patient arrives in some of these clinics, the first reception has to check his/her scheduled time.
- Patient is referred to a second reception.
- The medical examination is executed and the specialist creates a medical report.
- In the case of exams, the patient is first prepared. Then the exam is performed.
- The first reception has to provide information about date, time, addresses of appointments and exams. This step is called 'checkout'.
- It is possible that a specialist recommend for a specific patient a maximal priority.

Present and explain the DCR-Graph you presented to Eduardo

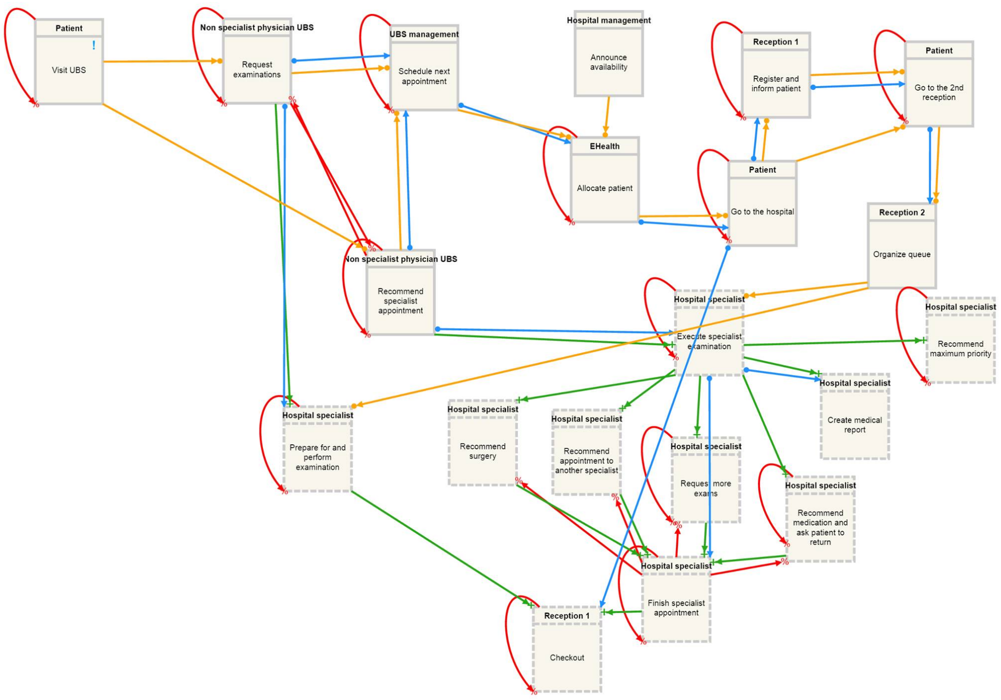


Figure 2.1 – The hospital DCR-graph that we presented to Eduardo. An updated big version can be seen in the appendix *figure 8.1*

It can be seen from the given DCR-graph that the only pending task, is that a patient initially visits the medical care unit UBS. Furthermore, at all times the hospital might announce availability of space to the E-health system. After the patient visits the clinic, the first part of the workflow is completed. At this point, we are presented with two mutually exclusive possibilities, of the patient either being sent for a more specialized examination, or being referred to see some specialist, both are performed by the physician at the UBS.

Say that the physician requests further examination after the patients visit. This initiates a new part of the workflow with two pending tasks. The first is the actual performance of the examination needed, according to the check at the UBS. This task will not be executable before the second pending task is executed, which is sequence of prerequisite administrative tasks. Once these pending tasks have been performed the "Checkout" task will be included and pending. Performing this task will finish the workflow.

On the other hand, say the UBS physician recommends a specialist for the patient, and therefore schedules an appointment. The condition for proceeding is that there has been announced availability at the hospital. This initiates a pending task which is for the specialist to perform the examination, meaning the examination must occur, but it is not available before multiple other administrative steps have been completed.

Once these self-excluding tasks have been executed the task of examination excludes itself, and a list of tasks are included in the process including additional surgery, checkups and medication. Two pending tasks are now available, being the opportunity to finish the appointment or write up the medical report. The request for additional exams and medication to take can be executed once, before it excludes itself, while surgery and referrals can be performed any given number of times, until the specialist is finished with the appointment, and the patient moves to check out. However, if the specialist has not written the medical report yet, this task will still be pending and the workflow will not be completed before this task has been dealt with, since that is required for this part of the patient's case to be considered closed.

Reversion

Our Skype meeting with Eduardo took place April 17 kl. 13:35. He provided feedback to our DCR-graph of the hospitals workflow, seen in *figure 2.1* and answered any questions we had about the requirements. The positive feedback he gave was that we were one of the only groups who used the response relationship and that we had made a good structure for the workflow.

As negative feedback he said that the creation of the medical report was semantic and the report would have been created before the patient visited the hospital and be updated on the fly. He was confused by the role "Patient" in our DCR-graph, because he did not understand how the patient could be part of the workflow *(perhaps he just didn't understand roles / the design used at dcrgraphs.net)*.

Doing the feedback, we had 3 questions:

- Who makes an examination? Is it a Hospital specialist or someone else?

  We asked this because it wasn't specified and yes it was the Hospital specialist.

- What file types are most used at the hospital?

  We had the idea of adding a data transfer feature but we didn't get a answer to that.

- Have you worked with DCR-graphs before?

  We liked to learn how global DCR-graph modeling worked and we did not need to ask because he appeared to be used to working with DCR-graphs.

Changes to our DCR-graph

The changes made from the feedback is:

- Added gate activities to minimise the use of include an exclude relationships, because thy do not stack with the other relationships (If a activity is excluded then it's condition relationships do not count).
- Removed the "Patient" role to remove the confusion, now the receptionist is responsible for executing the initial activities.
- Changed the "Create medical report" activity to "Update medical rapport" for more precision.

The changes can be seen in the updated DCR-graph in figure 2.2.

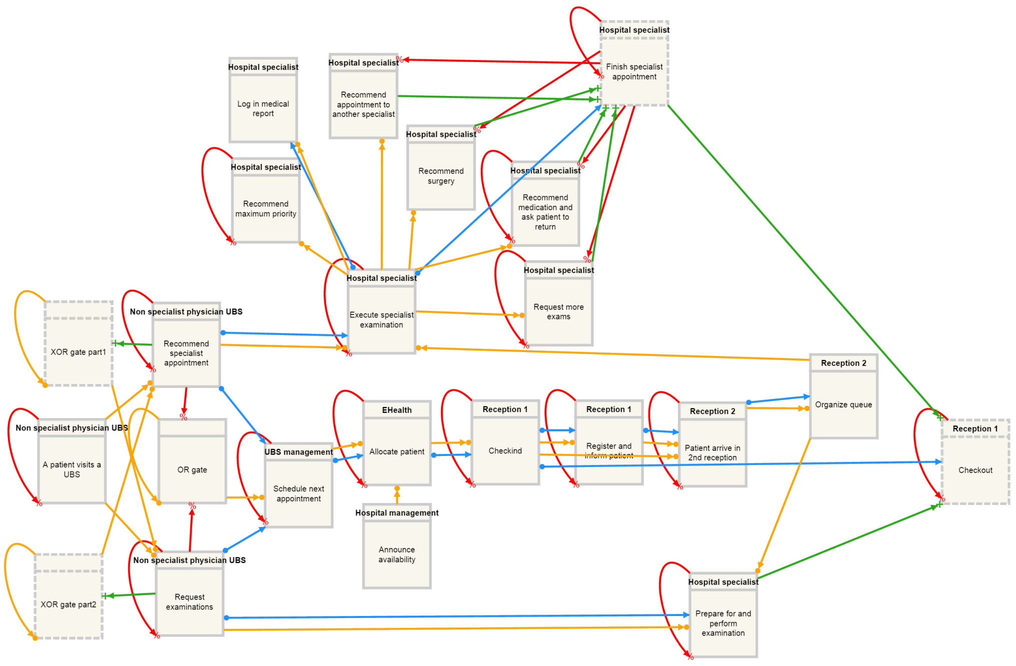


Figure 2.2 – The final version of the hospital DCR-graph, after feedback from Eduardo. *The gate activities can't be executed as they have them-self as condition, they are used because they use less relationships and stack better then Include/Exclude relationships.* A big version can be seen in the appendix Figure 8.2

2.2 Additional workflow example

We were given the workflow seen in figure 2.3, as an additional workflow to test our system on. Especial we should try to place the events on two different peers and execute them concurrently. The workflow is described in details in "Concurrency Asynchrony in Declarative Workflows"[3] Sec 2.1.

The events placements are controlled by a hash function (see section 3.3), therefore is it hard to control that two events are placed on different peers, but it is possible for a workflows events to be on multiple peers.

But it is not possible to execute multiple events concurrently, because of an error we did not have time to find.

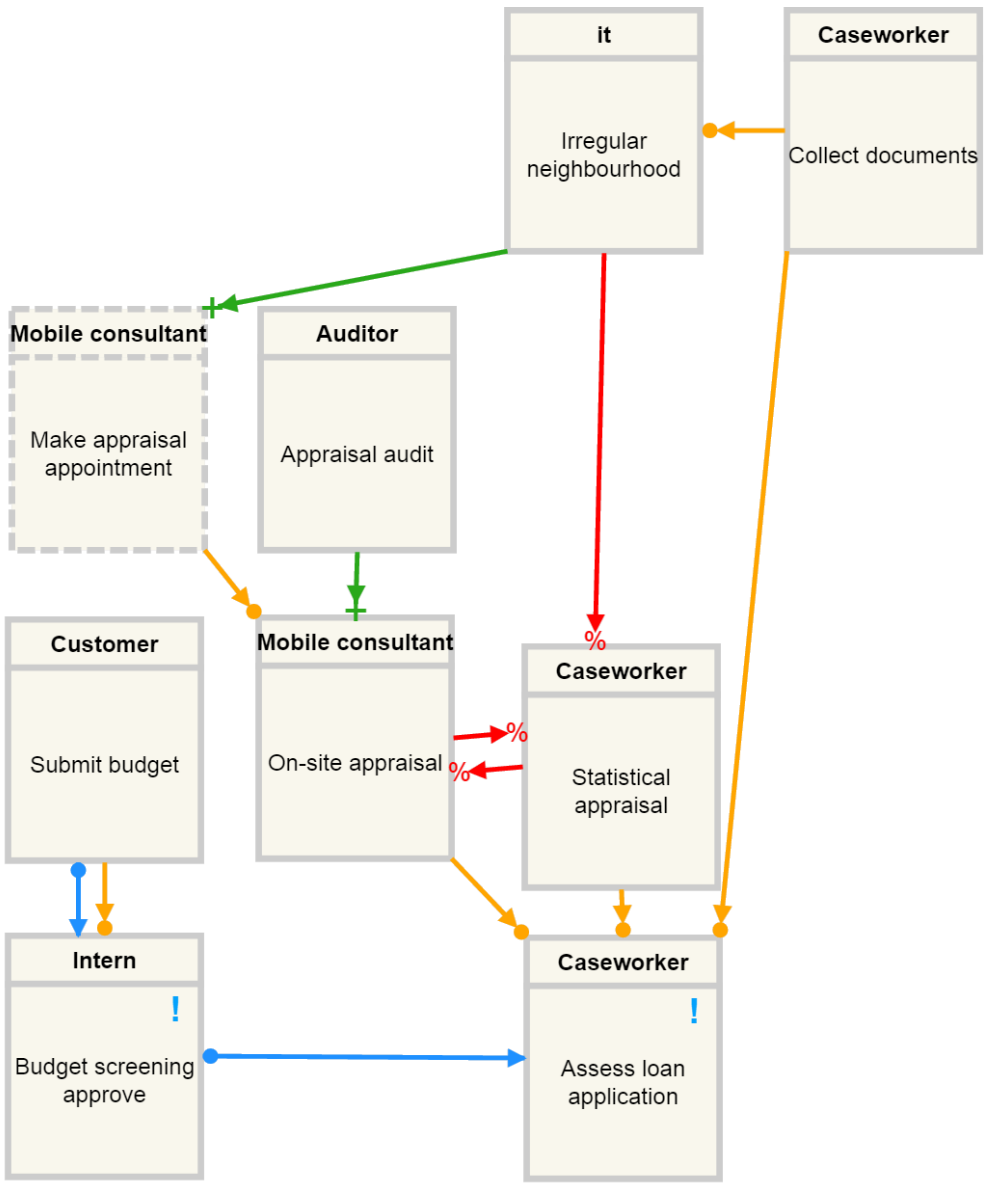


Figure 2.3 – The DCR-graph from "Concurrency Asynchrony in Declarative Workflows" [3]. *It describes a mortgage credit application workflow, from the financial services industry.*

CHAPTER 3

Description of implementation

3.1 General overview

Feature choices

By observing the list of unsupported features, it is easy to see that the implementation has had room for many improvements. Due to time constraints, we have focused on covering what we deemed the most important, for us to end up with a presentable project.

The supported features allows the program to run and manipulate the workflow, but there may still be issues with performance and correctness, and it might lack other features that would have been useful in a practical setting.

Implemented functionality

SERVER / Peer

- Persistency: Recovery of 1* node at a time failing/leaving the network (*a call has to try to route something through this dead node before a neighbor fails, for it to be noticed and recovered)
- Scalability: As many peers as you want (O(N) routing time, though)
- Scalability: Somewhat distributed* resource allocation (*our hashing is rather naive though)
- Scalability: Resource migration to a closer GUID
- Execution of events
- Relationtypes: Include, Exclude, Pending and Condition
- Initial event state (including, pending, executed, locked)

- Locking before execution
- Unlocking after execution
- Multiple user roles (for each workflow)
- Roles (for each event)
- Role based access control (check if the user has the right roles before executing)
- Logging of event execution (per workflow)
- Addition/removal of events
- Addition/removal of relations
- Multiple workflows* (*but with unique names eg. hospital1 and hospital2)

Not supported

- Actual good concurrency control
- User credential checks (currently a name is enough)
- Asynchronous http serving
- Timed pinging
- Easier server publishing (we have only tested on localhost)
- Pastry routing by routing table (better p2p scalability)
- Pastry locality utilization (better p2p scalability)
- Correct unwinding after failed event execution (events are not properly reset on failure)
- Addition and removal of roles for events
- Checks for RAM usage

MANAGER (simple setup manager)

- Read custom format text file and send creation requests

Not supported

- DCR-graphs XML without using our converter script

CLIENT

- Show workflow overview
- Change user
- Change workflow
- Execute event(s)
- Show logs

Not supported

- Change peer
- Get roles
- Show whether the workflow is completed (no pending)

Design

Design goals

High reliability and secure workflow execution flow
When modeling workflows in general, it is very important to ensure that events are completed in a particular order, and under the right conditions. As such, a high reliability for the execution of events, regarding concurrency and roles, is the ideal. The system should be able to keep a valid workflow state at all times, and ensure that only the users with right roles are allowed to execute an activity.

Flexible program structure and low coupling
Software during the course if its lifetime. As such, it is recommended practice to be prepared for replacing code or modules. As such it would be good to have low coupling to ensure that subsystems are easy to switch and less complex. Common interfaces should be used as much as possible to ease the process of adding new functionality.

Design choices

Using pastry for the distributed peer-to-peer serving
This seemed to be a good idea, since we already had some experience with the concept of this Distributed Hash Table, and seeing as it would make our implementation easier to work based on an existing system, rather than having to think up our own. In actuality the system ended up being some kind of hybrid, only really utilizing the notion of a leaf table, GUIDs and a ring-shaped node routing layout.

Using REST calls for the p2p routing
Convenient since we already had the experience and infrastructure for HTTP message sending and interpretation in place. No need to use custom checks since the library handles the protocol and ensures that most things are taken care of, which simplified our networking greatly.

Using F♯ for the entire program
We wanted more experience and it made it easier to make a more 'correct' program, by using matches and result types and completely avoiding mutability and null pointer issues. Few unexpected objects and casts too, and a very 'minimal' kind of application for better or worse. Definitely feels less bloated.

Not using any mutable data in the entire program
We choose not to use mutable data, so it would be easy-er to make the system multi-traded if we had time. But we do occasionally use C♯ interfaces and functionality in multiple places (as WebClient), but we do so only to save time.

Data management

The data management of our implementation uses functional programming and immutable data structures. Application state is constantly routed through functions and fields are modified as needed. In our recent design we have shifted away from using tuples and collections to represent our data, and more towards records, as these make for more considerations when unpacking record members, and makes it easier to only partially modify the input values before returning them.

Due to an early design choices based on time restrictions, we never considered storing the data in a persistent data format, so our solution uses RAM to store all the application data. This has the potential to cause loss of data on node failure, but as we handle this with our persistency, it should not cause larger issues. The biggest problem with this implementation is the limitation of storage space in RAM, as hard disk space is often much cheaper, but at the same time this ensures that data cannot be changed by another application or a malicious intruder, and that the application has much faster response since all data is "hot".

Our pastry routing module is completely agnostic of the data type of the application, since all pastry data structures are implemented to work with a generic type parameter <'a>, corresponding to whatever type of data the application needs to keep track of its internal state. This parameter is then passed through and returned in every interaction between the pastry layer and the application layer (using ResourceHandleFunc and SendFunc respectively), so that the data stays persistent. This also means that our distributed hash table system only ever keeps its data in RAM

Architecture

When designing software systems, it is a good idea first to reflect on existing architectures, as they might contain desirable properties and good abstractions, that might be usable in one's own architecture. For our peers we decided to create a layered architecture which would keep parts of the program interacting only in a fixed way, so as to both more easily delegate the development tasks, but also to make sure that one part of the program only takes care of one aspect of the functionality, in the general spirit of Object-Oriented Programming. Additionally, using a layered architecture would naturally promote low coupling, and more easily allow for incremental testing. On the other hand, such a composition might make it more difficult to add new functionality, and might incur a slight performance overhead, as the 'top' layer has to take more intermediate steps in order to reach the functionality at the bottom level.

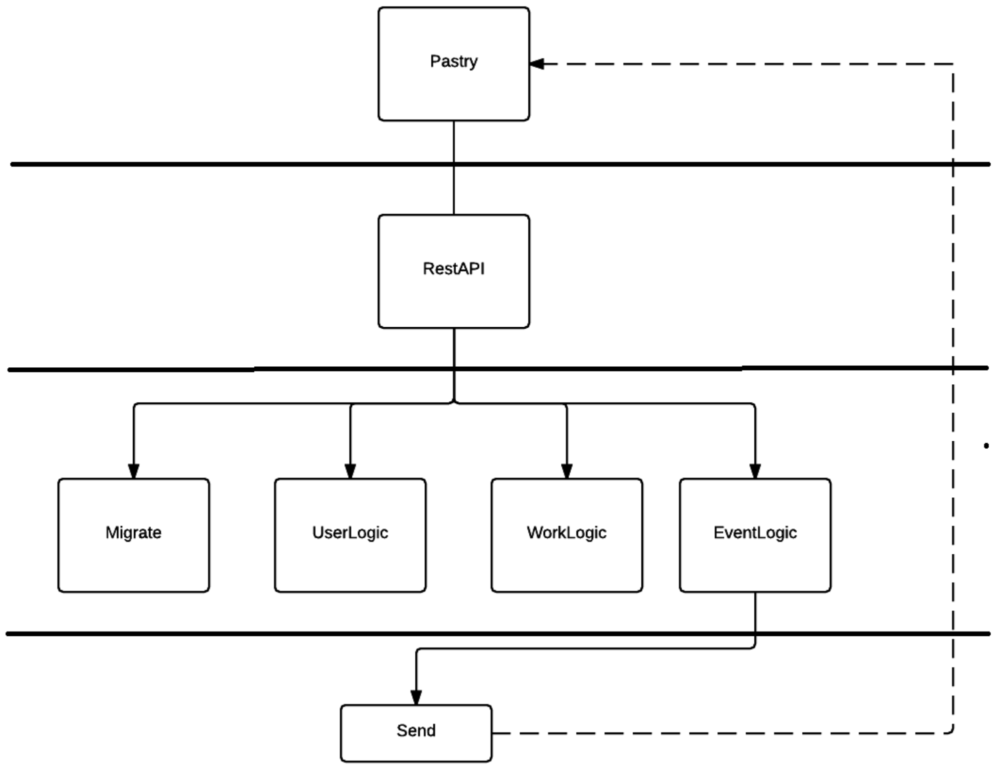


Figure 3.1 – Our layered architecture: Peers communicate at the top layer, but objects are only accessible/manageable though the bottom layer and those are only accessible though the RestAPI.

An issue that we encountered though the implementation was that EventLogic could have a case where it executed an event that had condition relation towards it from other events. In this case EventLogic would have to read the status of those events in order to execute the event. Because of this EventLogic has to ask Pastry to get the status of those events and as such we break the closed architecture we aimed for. It should however be said that Pastry is responsible for passing a send function down the layers, so in that sense Pastry is still responsible for the functionality of that Send function. The layer 'send' will make use of this Send when necessary. In a functional sense. What is actually happening is that Pastry defines a function that is passed down to EventLogic for use, but there is no actual reference to Pastry because everything is immutable. Ideally we would have had a reference from the workLogic class to this send function because if an workflow has to be deleted then it should issue a delete statement to Pastry for every event in the workflow. In our current implementation there is the issue that if the events are not all on the same peer then deletion of workflow won't work because EventLogic only deletes events on the peer and workflow makes use of EventLogic for that which was wrong (lack of time means this can't be fixed).

The low coupling we archieve by this architecture provides an easier way of dealing with security too. We know that any operation towards Users etc. is only done by UserLogic. This means we ensure that secure resources like this are only accessed from dedicated classes. Had it been done without this architecture we could have had a case where 2 modules manipolated secure resources and as such there would be two places to be concerned about the security. The low coupling ensures that if Users could be received without authority etc. then we would only need to make alterations to a single module.

3.2 Overview of modules

The main system is divided in multiple modules with individual responsibilities, as seen in Figure 3.2. But we also made a Client and a Manager program to make interaction with our main system easier. All the 3 programs have a commandline-based UI (the server's UI is not interactive, though).

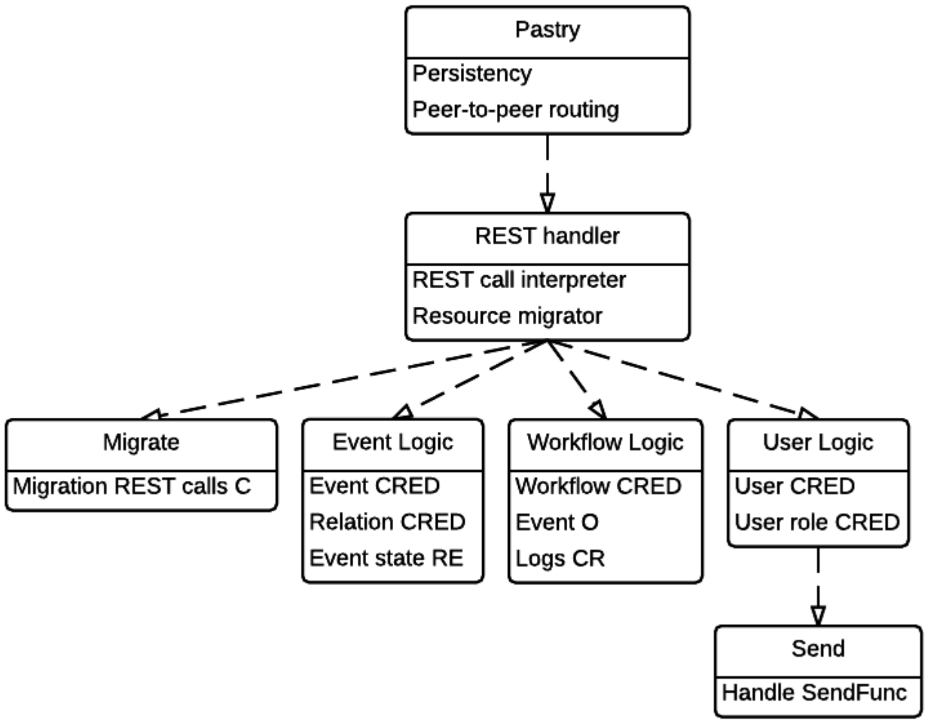


Figure 3.2 – The structure of a peer in our system: Pastry, is our top-level in a peer. It's here we get HTTP-requests, have date stored, and handle the pastry logic. If the HTTP-request has something to do with pastry it will be handled in the Pastry module. If that isn't the case, the request will passed further on to the API. Then the API module will translate the HTTP request and call the corresponding functions in one of the Event-, Workflow- or User modules.

Pastry is the module that by the use of Peer-to-Peer Pastry allows for events, workflows and users to be destributed as resources and will redirect a request to the right peer where a resource exist.

RestAPI When the proper peer has been found by Pastry, this module will process HTTP requests from the Client or an event asking for the statuses of others. RestAPI will then make use of underlying logic classes as appropriate to fulfill the request.

Migrate Is used to make a list of commands needed to restore / route (through the pastry) a given repository.

Logic modules Our resource objects events, users and workflows each has a logic class that allows for getting, updating, creating and deleting them. These classes also handle security matters such as if the authority is sufficient to execute an event.

Send Is a module that handles the case when logical classes needs to request something of Pastry. For EventLogic this could be an event that needed to know the eventstate of another event it has a relation to.

Client This is a software to ease the process for the end-user. It provides a console interface that asks the user what he wants to do and then requests a peer to execute the desire of the user.

Moderater Is used to send a DCR-graph to a peer, without having to write all the HTTP calls yourself. The DCR-graph is read from a *.txt* file.

3.3 Explanation of key parts

Pastry - Distributed Hash Table

Theoretical foundation

For our implementation of the distributed workflow system, we have chosen to be inspired by the Pastry distributed hash table as described in [4]. As we had initially gotten the impression that the system didn't have to be peer-to-peer, we only started adding this functionality late in the project, and thus have not attempted to make a full implementation of the Pastry DHT.

The system is more completely described in the aforementioned paper, but the gist is that:

- Every node has a Globally Unique ID (GUID)
- Every node knows the nodes with the |L|/2 smaller and |L|/2 GUIDs bigger than its own. These are called its 'leaves'
- Every node has a table of nodes, where the nodes in row 'n' shares 'n' 'digits' with the node's GUID, and column 'l' contains the GUID of a node whose deviating 'digit' is the 'l'th one. For instance, in a base 4 table of a node with id 1234, at the spot [2][2] would contain a node with the GUID 122X, since it shares 2 digits with the nodes, and thereafter has a '2'. This routing table ensures that in all except the rarest cases, a message sent to a route will get at least one digit closer to the node that it is being routed to, making the routing time for messages in the network logarithmic.

The reasons why we chose to implement this, is that it handles both persistency and scalability exceptionally, along with being distributed in a way that suits the requirements for this project. Additionally, it would simplify our implementation to use an existing architecture rather than inventing our own, and especially one as well-defined as Pastry.

Our implementation

Currently only the leaf-set part of Pastry has been implemented and is being used. This means that we currently only route through nodes in a given node's leaf set. This means that although we still will be able to route messages to all peers, once the amount of nodes exceeds the maximum size of the leaf set of each node (by each peer having all its closets leafs), our routing will start to take linear time. Normally Pastry would have a logarithmic routing time using its routing table, but we did not have time or incentive to implement this.

Persistency handling

In our program, the leaf set is also used for persistency, as each of the peers keeps a backup of the node to the right of it in the GUID space. Whenever a call for a resource in the network is made, we check whether the application state of the node handling has changed. If so, it sends a backup to the node to the left of it, made by serializing the state of the application. In order to make this work, we 'wrap' our GUID space, so that the otherwise normally 'smaller' leaf (to the left) might actually be a node with a higher GUID, if there are no suitable smaller leaves. Whenever a node tries to route something through a peer and gets a connection of failure, and routes a message towards the GUID of the dead node, informing the receiving node that the found note has died, and that the message should be handled or sent to the next neighbor, in order to find the node with the backup of the dead node. After this message has reached the node with the backup, this node then sends a special request to the application with the application state of the dead node, which the application then converts to the 'create' REST calls needed to restore this data to the network. This check on connections also means that nodes can just leave the network at any moment, as long as no two adjacent nodes goes missing at the same time (before one of them has been migrated).

Resource locations

In order to keep the location of our resources persistent, it is ensured that a resource can also be correctly located by finding the node with the GUID closest to the hashed event. We have to ensure that resources are properly positioned when a new peer joins the network. As a part of the Pastry DHT, whenever a node joins it will notify the peers in its leaf table that it has joined, so that they can also update their leaf tablesto properly include this new node. We take advantage of this by making a check on each leaf node at this point in time to see if the newly-joined peer is a neighbor of it, in which case it sends a message to the application to see what resources kept in it. If there are any, these resources must be relocated to the new node.

Peer-to-peer REST handling

The way that we handle resource requests, is that whenever a request using the ’resource’ URL prefix, we create a ’request’ object representing the requested resource (URL, HHTP method, data), and store the response channel in a list of channels for this request (to allow more requests for the same things). The resource request is then converted into a modified URL (using our ’pastry’ prefix for internal messages), which includes the address of the original peer who received the request. The modified message is then routed using unblocking internal messages, and once the resource has been found and handled on the correct node, the message is returned to the original recipient, which finally sends this reply through its response channels. This way of doing it ensures that we only block the client sending the original resource REST call, while the rest of the nodes needed during routing are free to handle other requests, as soon as they have passed the message along.

Communication between the routing and application logic layers

It was the intention to make sure that the routing module was as loosely coupled to the application logic as possible. In order to do this, when one starts the routing server they also need so supply several callbacks, which the routing uses to handle requests for resources in the system.

This function is what we call a "Resource Handling Function" (the ResourceHandleFunc type), and allows for the routing logic to send messages to the application. For this communication to also be possible the other way, as when a workflow activity for instance needs to see/modify the state of another activity, we pass in a reference to a function (SendFunc). These can then be used to route requests from inside the application and through the routing layer. In order to keep our state consistent during the constant updates, these functions pass the state of both the p2p node (Node), and the application data (Repository) through (using PastryState). The reasons why it is necessary to pass the node’s data through this, is due to that we might get trouble otherwise, if the node found a dead peer when routing on behalf of the application and updated itself, and this change was not kept once this was finished. This routing function is then used to both find resources located on the same node as well as on others, as it forces the application to either send the request back through the resource handling function, or transforms the node into a ’client’ requesting a resource from a peer elsewhere.

Resource Manager (RestAPI)

The resource manager processes the HTTP resource requests sent through the routing layer, and interprets these calls, then delegates them to the logic classes such as EventLogic, UserLogic and WorkflowLogic, described in 3.3. In our appliaction, it contains the 'Resource Handling Function' mentioned in 3.3.

REST API

Below is the complete list of possible requests to make to the REST API of our application. this covers the HTTP Verbs POST/GET/PUT/DELETE, corresponding to the REST commands "Create", "Read", "Edit/Update" and "Delete".

All of the shown URLs are to be prefixed by the address of the peer you are contacting and a "resource" tag, denoting that you are accessing a resource. This is because we also use REST for our peer-to-peer interaction, which uses the prefix "pastry".

An example would be to use GET "http://127.0.0.1:80/resource/" + "user/Jakob/roles/study" to get the roles of the user "Jakob" for the workflow "study" from the known peer "127.0.0.1:80".

1. Add a user

   POST user/<name>

2. Delete a user

   DELETE user/<name>

3. Get the roles that a user has for a given workflow

   GET user/<name>/roles/<workflow>

4. Add the given user roles to the user's permissions for this workflow

   PUT user/<name>/roles/<workflow>/<event>

   <role1>,<role2>,...<roleN>

5. Remove the given user roles from the user's permissions for this workflow

   DELETE user/<name>/workflow/<name>

   <role1>,<role2>,...<roleN>

6. Create a new workflow

   POST workflow/<name>

7. Get the list of events in the given workflow

   GET workflow/<name>

8. Add an event to the list of events for a workflow

   PUT workflow/<name>

   <eventname>

9. Delete a workflow

   DELETE workflow/<name>

10. Create new event with the given state and roles

    POST workflow/<workflow>/<event>

    [0 | 1][0 | 1][0 | 1][0 | 1],<role1>,<role2>,...<roleN>

11. Get the state of an event attribute

    GET workflow/<workflow>/<event>/included

    GET workflow/<workflow>/<event>/pending

    GET workflow/<workflow>/<event>/executed

12. Set the state of an event attribute

    PUT workflow/<workflow>/<event>/included

    PUT workflow/<workflow>/<event>/pending

    data: true | false

13. Delete an event (currently this only updates the data structure, and not the list in the workflow)

    DELETE workflow/<workflow>/<event>

14. UNSUPPORTED Add roles to an event

    PUT workflow/<workflow>/<event>/roles

    <role1>,<role2>,...<roleN>

15. UNSUPPORTED Remove roles from an event

    DELETE workflow/<workflow>/<event>/roles

    <role1>,<role2>,...<roleN>

16. Try to execute this event as a user

    PUT workflow/<workflow>/<event>/executed

    <user>

17. Get whether the given user can execute this event (checks both permissions and state)

    GET workflow/<workflow>/<event>/executable?data=<user>

18. Lock the event or fail if it is already locked

    PUT workflow/<workflow>/<event>/lock

19. Unlock the event or fail if it is not locked

    PUT workflow/<workflow>/<event>/unlock

20. Check if outgoing condition relations from this event are met (the event is excluded or executed)

    GET workflow/<workflow>/<event>/getif

21. Add a relation from this event to another (used for exclusion/response/inclusion when executing)

    PUT workflow/<workflow>/<event>/exclusion/to

    PUT workflow/<workflow>/<event>/condition/to

    PUT workflow/<workflow>/<event>/response/to

    PUT workflow/<workflow>/<event>/inclusion/to

    data: <eventname>

22. Add a relation from another event to this one (used for checking conditions when executing)

    PUT workflow/<workflow>/<event>/exclusion/from

    PUT workflow/<workflow>/<event>/condition/from

    PUT workflow/<workflow>/<event>/response/from

    PUT workflow/<workflow>/<event>/inclusion/from

    data: <eventname>

23. Remove a relation

    DELETE workflow/<workflow>/<event>/<relationtype>/to | from

    <eventname>

24. Create a new execution log entry

    PUT log/<workflow>/<event>

    <date>,<username>

25. Get the execution logs of a workflow

    GET log/<workflow>

The logic modules

The architecture chapter 3.1 describes why we chose to use logic classes, namely to make as low coupling as possible in our system by using a layered architecture. The 'logic' classes are all designed with our REST protocol in mind, as they each contain the methods necessary for the CREATE, READ, EDIT and DELETE that our API should support. All classes are designed to return a proper type value depending on the result. For the eventlogic this would be OK, Unauthorized, NotExecutable (when trying to execute), MissingRelation (if you try to delete a non-existant relation), LockConflict if an event is locked when you try to execute it etc. and ERROR if a crash etc. occur. The other types has similar type values as results (workflowlogic can return MissingWorkflow etc.).

To simplify the process of sending requests for other resources in the network, the module named Send was created. This module wraps the 'SendFunc' provided by the routing layer (see: 3.3), in a way that makes it easier to make the correct calls, by utilizing algebraic data types to limit the potential for errors.

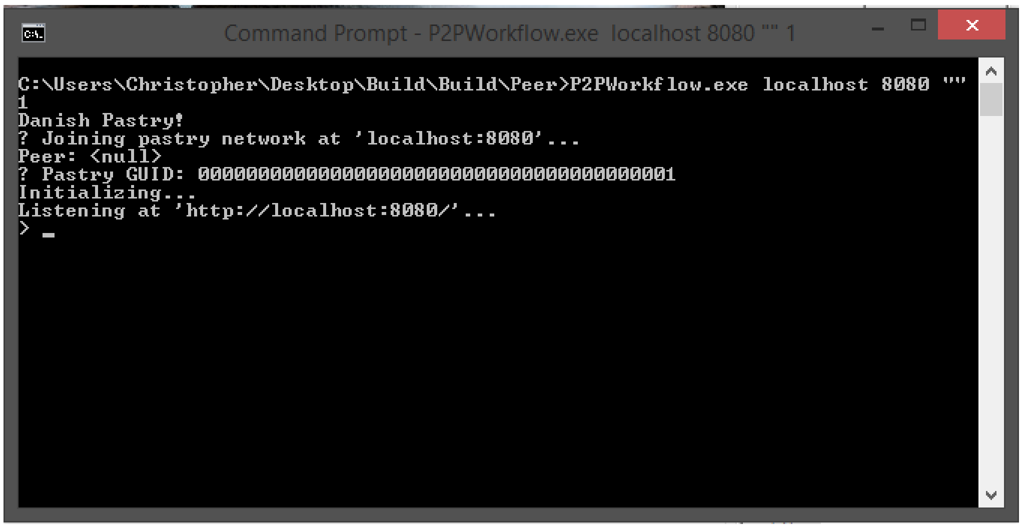
Client

Our client is a simple commandline-based program which uses our application's REST API to present the user with the state of a workflow, and shows them what actions are available to take on the workflow. The client allows the user to set which workflow they are working with, and which user name to use. In addition to this, the user can choose to view the state of the workflow, and be presented with and choose which executable events in the workflow to execute. It is the way in which we've fulfilled the requirement of having some kind of client UI to interact with our system.

3.4 Illustrative examples

User guide

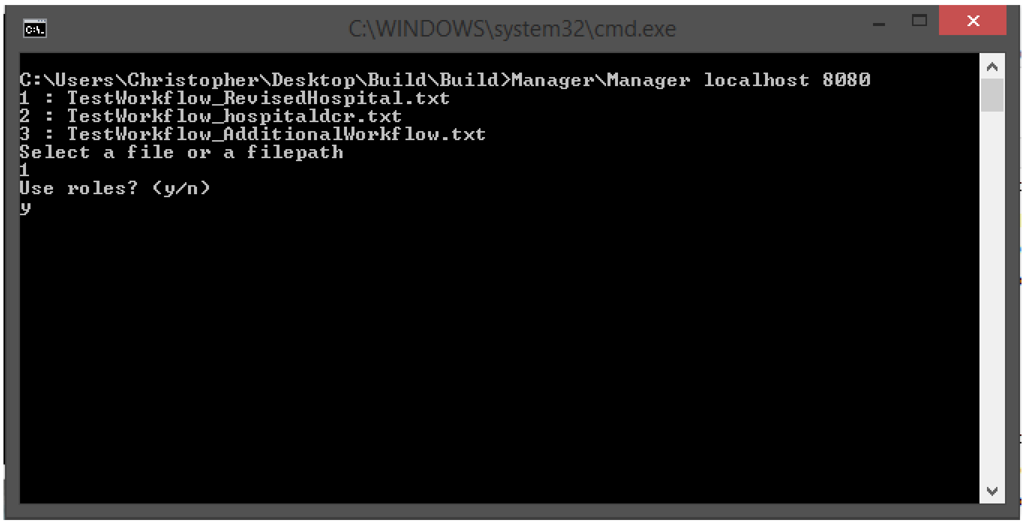
User guide - Setting up the program Initially the P2PWorkflow, located in RESTfulWorkflow/Peer, is compiled and executed through the command line. This is done by giving multiple arguments in the form of: "address" "port" "peer_address_with_port" and the "GUID". Upon starting up the program for the first time, only the two first are required for the program to work, as seen in figure 3.3.



```
C:\Users\Christopher\Desktop\Build\Build\Peer>P2PWorkflow.exe localhost 8080 ""
1
Danish Pastry!
? Joining pastry network at 'localhost:8080'...
Peer: <null>
? Pastry GUID: 0000000000000000000000000000000000000001
Initializing...
Listening at 'http://localhost:8080/'...
> _
```

Figure 3.3 – Picture guide of how to run the P2PWorkflow.exe.

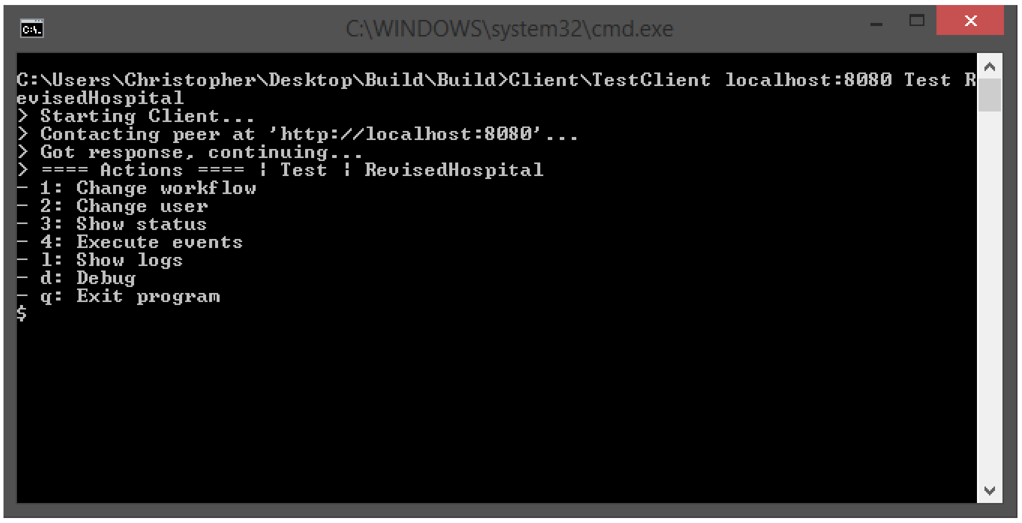
Next, compile and run the manager (setup program) located in “RESTfulWorkflow/Manager”. This is again done through the command line, and we are now prompted with a list of workflows to be chosen between. Press the corresponding number for the workflow you wish to work with. This can be seen in figure 3.4.



```
C:\Users\Christopher\Desktop\Build\Build>Manager\Manager localhost 8080
1 : TestWorkflow_RevisedHospital.txt
2 : TestWorkflow_hospitaldcr.txt
3 : TestWorkflow_AdditionalWorkflow.txt
Select a file or a filepath
1
Use roles? (y/n)
y
```

Figure 3.4 – Picture guide of how to run the Manager.exe.

This automatically compiles the workflow server in 'Event' and runs it. Please note that the P2PWorkflow server window must be kept open for the client program to be able to communicate with it. Next you will be prompted to decide whether you want to use roles in the workflow. If you choose to use roles, the execution of events will require the correct role. Since multiple roles are not supported by our client, you will most likely not be able to finish a workflow if the workflow uses more than one role. Compile and run the client located in "'RESTfulWorkflow/Client", and you are ready to start. In the commandline we are prompted with three required arguments: "peer_address" and "user" "workflow". Connect to the P2PWorkflow as seen in figure 3.5 and continue by following the manual below.



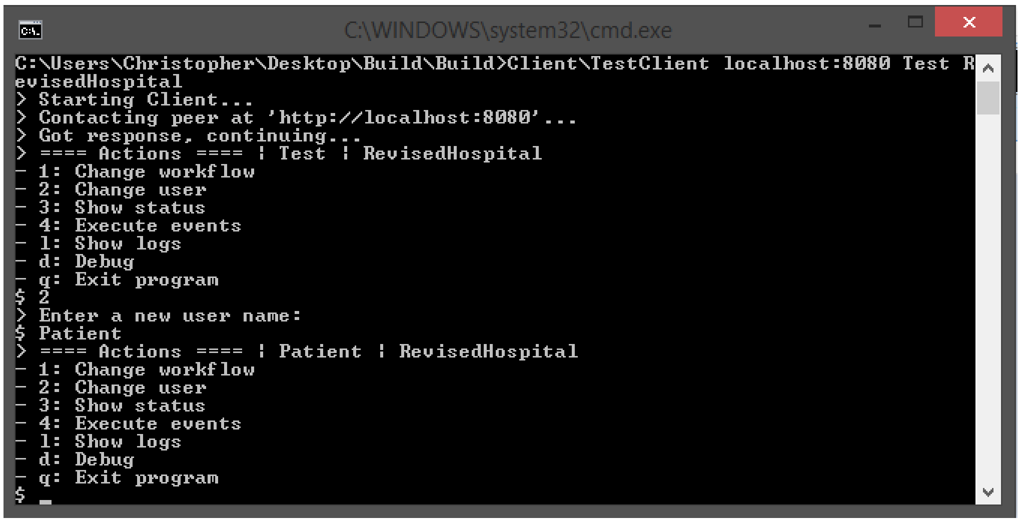
```
C:\Users\Christopher\Desktop\Build\Build>Client\TestClient localhost:8080 Test R
evisedHospital
> Starting Client...
> Contacting peer at 'http://localhost:8080'...
> Got response, continuing...
> ==== Actions ==== | Test | RevisedHospital
- 1: Change workflow
- 2: Change user
- 3: Show status
- 4: Execute events
- l: Show logs
- d: Debug
- q: Exit program
$
```

Figure 3.5 – Picture guide of how to run the TestClient.exe.

User guide - Client manual In figure 3.5 we see the initial view of the Client. After we connect to the P2PWorkflow, we are now presented with a number of options. First off, we can change the workflow. Secondly we can change the user (role), which only works in the role-based client. Furthermore we have the possibility to show the status of the connection. By doing this we will be presented with all events. This is shown in a similar fashion in the next step, the only difference being that an arrow is shown beside the events which are ready for execution. By pressing "l" it is possible to show the logs for the workflow. This helps backtrack the previous steps taken. By pressing "d" we can debug, and potentially discover what might be going wrong in the program. Finally we can press "q" to exit the program.

Example runs

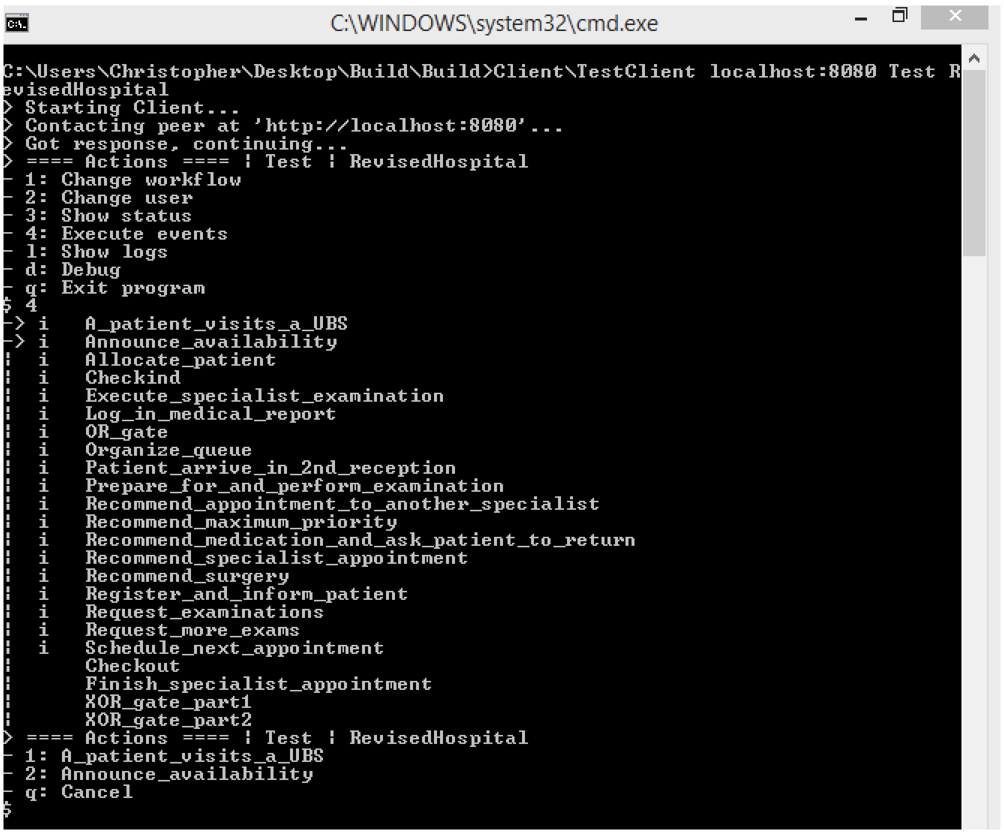
Illustrative example runs From the initial view of the client explained in the previous section, we will now proceed to an example run. Since the workflow and role is already chosen, we could immediately proceed to the workflow. Consider the possibility that an incorrect role is chosen. We easily switch this by pressing "2" and entering the name of the desired role. These actions are shown in figure 3.6.



```
C:\Users\Christopher\Desktop\Build\Build>Client\TestClient localhost:8080 Test R
evisedHospital
> Starting Client...
> Contacting peer at 'http://localhost:8080'...
> Got response, continuing...
> ==== Actions ==== ¦ Test ¦ RevisedHospital
- 1: Change workflow
- 2: Change user
- 3: Show status
- 4: Execute events
- l: Show logs
- d: Debug
- q: Exit program
$ 2
> Enter a new user name:
$ Patient
> ==== Actions ==== ¦ Patient ¦ RevisedHospital
- 1: Change workflow
- 2: Change user
- 3: Show status
- 4: Execute events
- l: Show logs
- d: Debug
- q: Exit program
$ _
```

Figure 3.6 – Here a role switch is shown, with the corresponding actions and arguments.

By pressing 4, we are presented with a printed list of events. The ones marked with an arrow are the ones that are possible to execute, which can be seen in figure 3.7.

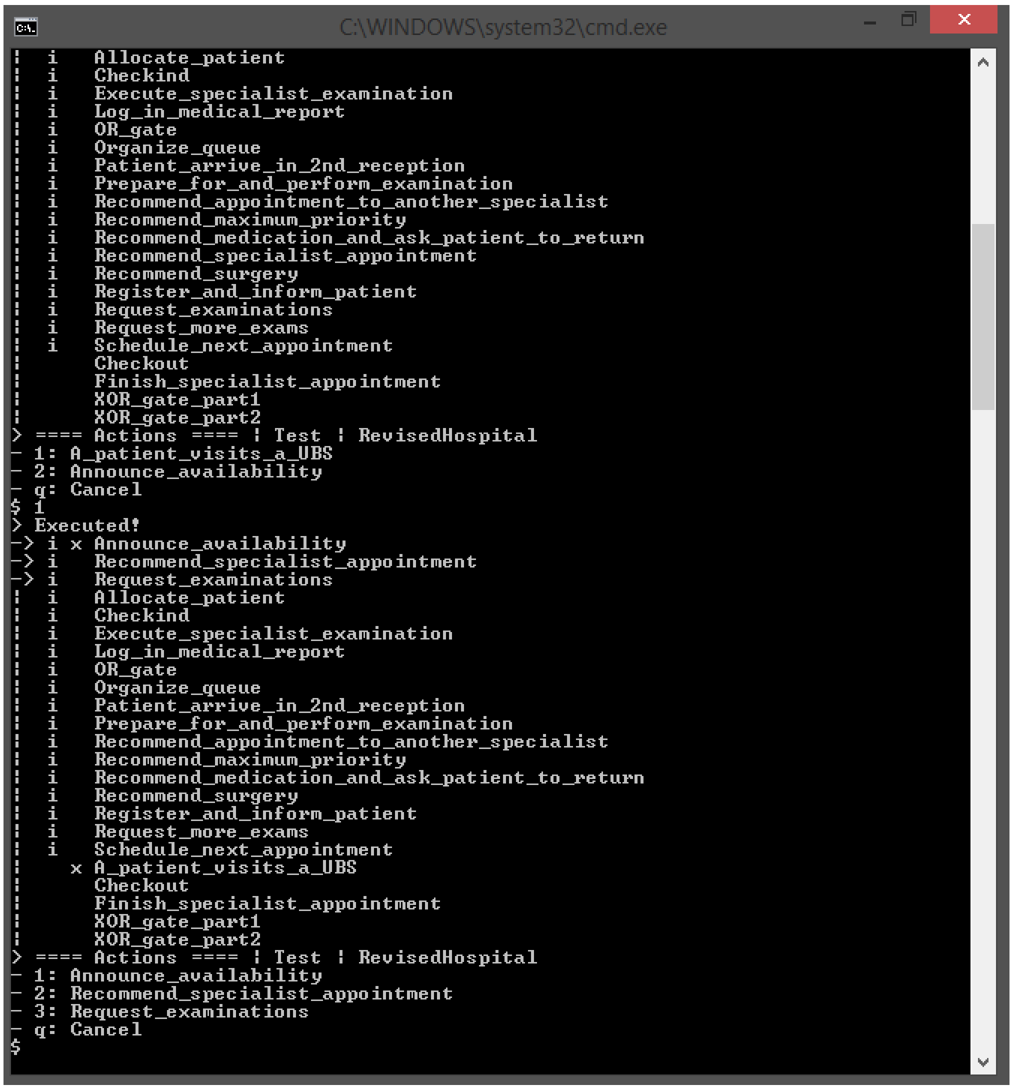


C:\WINDOWS\system32\cmd.exe

```
C:\Users\Christopher\Desktop\Build\Build>Client\TestClient localhost:8080 Test R
evisedHospital
> Starting Client...
> Contacting peer at 'http://localhost:8080'...
> Got response, continuing...
> ==== Actions ==== ¦ Test ¦ RevisedHospital
- 1: Change workflow
- 2: Change user
- 3: Show status
- 4: Execute events
- l: Show logs
- d: Debug
- q: Exit program
$ 4
-> i   A_patient_visits_a_UBS
-> i   Announce_availability
¦  i   Allocate_patient
¦  i   Checkind
¦  i   Execute_specialist_examination
¦  i   Log_in_medical_report
¦  i   OR_gate
¦  i   Organize_queue
¦  i   Patient_arrive_in_2nd_reception
¦  i   Prepare_for_and_perform_examination
¦  i   Recommend_appointment_to_another_specialist
¦  i   Recommend_maximum_priority
¦  i   Recommend_medication_and_ask_patient_to_return
¦  i   Recommend_specialist_appointment
¦  i   Recommend_surgery
¦  i   Register_and_inform_patient
¦  i   Request_examinations
¦  i   Request_more_exams
¦  i   Schedule_next_appointment
¦      Checkout
¦      Finish_specialist_appointment
¦      XOR_gate_part1
¦      XOR_gate_part2
> ==== Actions ==== ¦ Test ¦ RevisedHospital
- 1: A_patient_visits_a_UBS
- 2: Announce_availability
- q: Cancel
$
```

Figure 3.7 – A printed list of the events available, with the events pending for execution are marked with an arrow.

Furthermore, a list of the available events are printed at the bottom of the screen, each marked by a number enabling them to be executed. After an event is executed, the list of available events is updated and new possible events are presented. It can also be viewed from the list, which of the events that were executed as they are marked with an "x" shown in figure 3.8.



```
¦  i   Allocate_patient
¦  i   Checkind
¦  i   Execute_specialist_examination
¦  i   Log_in_medical_report
¦  i   OR_gate
¦  i   Organize_queue
¦  i   Patient_arrive_in_2nd_reception
¦  i   Prepare_for_and_perform_examination
¦  i   Recommend_appointment_to_another_specialist
¦  i   Recommend_maximum_priority
¦  i   Recommend_medication_and_ask_patient_to_return
¦  i   Recommend_specialist_appointment
¦  i   Recommend_surgery
¦  i   Register_and_inform_patient
¦  i   Request_examinations
¦  i   Request_more_exams
¦  i   Schedule_next_appointment
¦      Checkout
¦      Finish_specialist_appointment
¦      XOR_gate_part1
¦      XOR_gate_part2
> ==== Actions ==== ¦ Test ¦ RevisedHospital
- 1: A_patient_visits_a_UBS
- 2: Announce_availability
- q: Cancel
$ 1
> Executed!
-> i x Announce_availability
-> i   Recommend_specialist_appointment
-> i   Request_examinations
¦  i   Allocate_patient
¦  i   Checkind
¦  i   Execute_specialist_examination
¦  i   Log_in_medical_report
¦  i   OR_gate
¦  i   Organize_queue
¦  i   Patient_arrive_in_2nd_reception
¦  i   Prepare_for_and_perform_examination
¦  i   Recommend_appointment_to_another_specialist
¦  i   Recommend_maximum_priority
¦  i   Recommend_medication_and_ask_patient_to_return
¦  i   Recommend_surgery
¦  i   Register_and_inform_patient
¦  i   Request_more_exams
¦  i   Schedule_next_appointment
¦    x A_patient_visits_a_UBS
¦      Checkout
¦      Finish_specialist_appointment
¦      XOR_gate_part1
¦      XOR_gate_part2
> ==== Actions ==== ¦ Test ¦ RevisedHospital
- 1: Announce_availability
- 2: Recommend_specialist_appointment
- 3: Request_examinations
- q: Cancel
$
```

Figure 3.8 – An updated list of events, after a few events are executed. Events that are excluded after execution can no longer be viewed on the list, and events that are unlocked now show up.

After traversing through the DCR-graph, we eventually reach an ending. Upon completing the workflow or cancelling the workflow, the program can be terminated by pressing "q" returning to the state shown on figure 3.5 , and another can be commenced if desired.

Chapter **4**

Testing

4.1 Overall Test Approach

Our initial plan was to black- and white-box component testing of essential parts of both the program and our Client, integration testing of the program and system testing of all the functional requirements. But all testing was cut when we were given the second part project, because of time pressure. This chapter will mostly describe the testing which is out of date, except system testing there will describe our last attempt at making sure all functional requirements are met.

All Unit tests was made using F♯ because it made the tests more manageable and readable. We used the NUnit unit testing framework to test our program, as multiple group members are familiar with the framework.

Testing strategy

As seen in *diagram 4.1*, the program was separated in multiple modules there was to be tested separately and together. We planned to make black box unit test for every module and integration test for the peer. In the first part of the project there was no plan for intern peer testing, as we thought that the program was going to be client server structure. B was unnecessary for the final product, but was also unit tested as we thought we had the time.

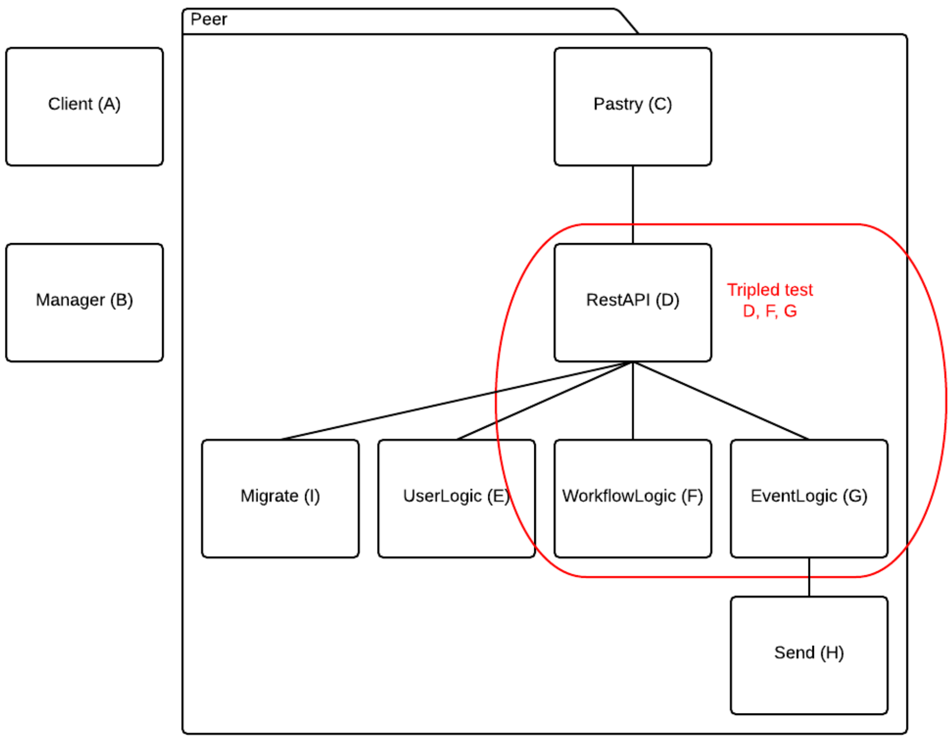


Figure 4.1 – Our module structure: the peer is our main program and was to receive focus under testing, the tripled integration test D, F, G was made before C, H and I was implemented or a part of the design. B was unnecessary for the final product.

4.2 Component / Unit Testing

In the second part of the project did we try to make unit tests for G from *diagram 4.1*, to make sure it worked even as D was not updated to the new system specifications yet. But because of late design changes and functionality cuts to save time, all the unit tests are unusable at this point, the unit test can be seen in the appendix. The test plan can be seen in *table 4.1*.

C was tested with constantly debugging with and without multiple peers to make sure that the Pastry functionality worked as described in [4], these tests was never written down as we were unsure of the architecture and did not have time to rewrite the tests each time we read [4].

function name	**Input**	**Output**
create_event	Name & State	A new event
create_event	Name & State	A new event
check_roles	the right role	true
check_roles	the wrong role	false
check_roles	role less	true
check_roles	same role twice	true
execute	the right state	true
execute	the wrong state	false
add_event_roles	no roles	true
add_event_roles	same role twice	true
remove_relation_to	a role	true
remove_relation_to	same role twice	true

Table 4.1 – The unit tests made for G module

4.3 Integration Testing

The tripled integration test D, F, G from *diagram 4.1* was made in the first part of the project as a black box test and mainly tested two points, whether the state of the workflow is correct after various operations and whether the debug functionality works correctly (The test can be seen in the appendix). These test was for the old design and is unusable in the new, but it taught us a lot about how DCR-graph works, because the test was written independently from the implementation, so it showed where it was easy to get misconceptions about how DCR-graphs works.

4.4 System testing

In the last stage of development we made system testing by setting one of our best programmers to go through all the functional requirements seen in chapter 1.1 and all modules showed in diagram *4.1*, to make sure we have sovled them and fix any failures there could hinder essential functionality.

CHAPTER 5

Role Based Access Control

5.1 Approach to implementing RBAC

Role based access control is the task of controlling access to shared resources[5]. If a resource is not shared then it could simply be isolated and thereby made secure. Our access control is however a multiuser system where each user has a name and role. A role gives specific rights, but our system uses a very optimistic approach that assumes there is only one user with a specific name. It also doesn't use any passwords or other authentication. The only restriction is that you cannot execute an event unless the user you provide has the correct roles. However, you can manually set the state of the events if you so desire (included, pending). The latter makes our system very vulnerable to security attacks as no role is required for tampering with any instruction aside from execute. The reason this only exists for execute is that our access control has only been implemented from a DCR-graph perspective (who can execute what and when). A very common attack is altering packets that has been caught by a sniffer etc. In this case it doesn't take a genius to figure out how to change the HTTP verb from GET to DELETE when eventname is clearly present in the URL of the HTTP request.

As such our system suffers from the following 3 classes of security.

- Leakage : Recipients can receive resources they are unauthorized for.

  GET can be issued without supplied role on users, events and workflows.

- Tampering : Resources can be altered without authorization

  A DELETE statement etc. could be issued without supplied role.

- Vandalism : Interference with the proper operation of the system.

A fake pastry instruction can be send to a peer to change the pastry state of a peer.

Our hospital workflow from sec 1.3, has the following roles and may execute the given activities if their conditions are satisfied by the DCR workflow.

- Hospital Specialist

  Execute specialist examination

  Log in medical report

  Recommend appointment to another specialist

  Recommend surgery

  Recommend medication and ask patient to return

  Request more exams

  Recommend maximum priority

  Finish specialist appointment

  Prepare for and perform examination.
- Non Specialist Physician UBS

  A patient visits a UBS

  Recommend specialist appointment

  Request examinations
- UBS Management

  Schedule next appointment
- Hospital management

  Announce availability
- EHealt

  Allocate patient
- Reception 1

  Checkin

  Register and inform patient

  Checkout

- Reception 2

    Patient arrive in 2nd reception

    Organize queue

The actors or roles above are the possible roles that each have the authority to execute different kinds of activities. A user can be assigned to one or more of these. Given that we are using a layered architecture where the event, workflow and user objects are handled by their own classes, we are ensuring that any security issue related to them should primarily be the responsibility of their respective classes. For this we have EventLogic, userLogic and workflowLogic. These logic classes are accessible by a common interface that does not care about security. This in itself is another security concern (exposed interfaces).

On the following access control matrix we can see some possible actors. Pastry is the first actor and has exclusive rights for exchanging/manipulating what we in this particular case call PastryState object. This consists of all Pastry related HTTP instructions sent between peers. Then we have the client without software. This one is the severest security hole we have that we would have fixed had been time for it. The thing is that aside from executing events, all other objects can be tampered with regardless of supplied roles using a browser. This makes our system very vulnerable to security attacks.

Aside from all this, our system will suffer less damage from denial of service attacks when compared to a standard client-server architecture. The fact that our system is peer to peer and can survive crashes (for the most part) means that flooding a single peer won't necessarily affect the other peers. Of course this would cause a partial denial of service, but it is a positive outcome, that the amount of potentially affected users are reduced.

5.2 Discussion of the Role Based Access Control

Access control is a serious concern that should be addressed when talking about security. Our system has documented flaws in this subject, but we know how it could be fixed. Should we have fixed our issues with users without passwords, providing identity on every request and issuing instructions without authority, we could have checked for the authority of the sender in every case. A client should ideally have received an identity

Objects Actors	Event	User	Workflow	PastryState
Pastry				«POST» «GET» «PUT»
Client without software	«POST» «PUT*» «GET» «DELETE»	«POST» «PUT» «GET> «DELETE»	«POST» «PUT» «GET» «DELETE»	«POST» «PUT» «GET» «DELETE»
Client with role using software	«PUT» «GET»		«GET»	
Manager	«POST»< <PUT» «GET» «DELETE»	«POST» «PUT» «GET» «DELETE»	«POST» «PUT» «GET» «DELETE»	

Table 5.1 –

Access matrix for possible actors of the system both internally and externally.

token after having gotten the hashed username and password verified, and that token should have been used as a verifier of the user's identity and authority. Storing access tokens would simply be another resource in the pastry network, which shouldn't have been a problem.

An issue we worked with was a natural occurring situation of vandalism, where the DCR-graph could be violated. The concept is that if an activity has to be executed, but has a condition to an excluded activity. In the case that another event was executed prior and issued an include to the previously excluded activity, then the first activity should not longer be allowed to be executed. Doing that would execute the first activity, but leaving its condition unsatisfied. To fix this problem we simply added locks on all events where it would be required to lock the conditions before executing a command. Doing this would ensure that the first activity wouldn't be able to execute because the condition had been locked and thus making it necessary to wait for it to be unlocked before proceeding. We call this vandalism because we are messing with the proper operation of the system.

CHAPTER **6**

Concurrency Control

6.1 Concurrency problems

Concurrency control

For our program, we chose rather naively to use synchronous and blocking HTTP calls for our communication. While this greatly simplified the implementation, it kept with it a fair share of problems. One of the benefits of this model was, however, that we would never get a wrong transaction or wrong data state, since all our data was immutable, and only one request would be handled at a time. This ensured that our program, unless it deadlocked, would always have a valid state with no transaction violation.

Example

One of the problems we had with our implementation, had to do with the way that we implemented persistency. We only have a very simple persistency model, in which only a single node can fail at a time, in order to not lose data in our distributed hash table.

This problem had to do with the fact that our servers would take a backup every time they passed a resource request through them, which meant that if a request for "user/Stinus" went through peers ′10′ and ′20′, both ′10′ and ′20′ would tell the node to the left of itself in the GUID space (eg: with nodes 10, 20, and 30, 20 would send this to 10), to keep a sent backup state for it, in case of crashes.

Due to this behaviour, we effectively got a deadlock when routing a resource through two peers, in a network with only those to in it, since 10 tried to send a message to 20, while 20 at the same time sent to 10, with both using blocking message requests.

To alleviate this, we inserted a check for whether the state of the node that routed the message ′actually′ changed, which meant that at most the one node who actually received the resource request would trigger the backup service.

6.2 Discussion of concurrency control

Concurrency control

Since our system is networked, we have naturally had to include some sort of concurrency control. This is a concern since we want our system to allow multiple users to access different peers, while interconnected workflow activities can be stored on different peers. Since we are working with workflows it is essential that our system operates independently of interference from operations performed on behalf other concurrent clients[5]. It is of course also a concern in situations like this, that peers do not reach deadlocks with other peers.

The concurrency control of our system is rather simple. We chose to use blocking HTTP calls for all communication between peers. Due to this fact, we mostly had problems to do with deadlocks (as we do not properly use timeouts for our requests), as our single-threaded blocking nature meant that data races were not a possibility.

We use locking to ensure that relations to something being executed are be locked before an execution can find place. This means that an execution request will cancel before updating anything, if it fails to lock all of the implicated activities first, and then proceed to unlock them again. This ensures that execution is a transaction. This is also known as the atomicity aspect "All or nothing", as either all parts of the change is made, or none of them. Another aspect that should hold in a good concurrent system is 'isolation', which entails that each request must be carried out without interference from other requests. This makes sense since the only situation in which the effects must be visible is when the request has been successfully executed. As we only ever handle one request at a time on each peer, this 'isolation' is easily achieved.

Deadlocks

Whenever a HTTP request is sent, then due to the way the system sends HTTP calls, we had to wait for a response. For clients, this waiting would likely pose no problem, but for a peer, a deadlock might occur. If we observe figure 6.1 we can see that if peer 1 received a request to execute

'Visit', then it would send the request to Peer 2. The activity 'Visit', however, requires that 'Checkin' has been executed, and therefore Peer 2 will have to send a request to peer 1, asking if the activity 'Checkin' had been executed or excluded. If Peer 1 was occupied waiting for a response from the 'Visit' execution request, then it won't ever be able to answer the request from peer 2, which means that none of them can proceed before receiving an answer from the other, which is a phenomenon called a 'deadlock'. For this reason peers should reply instantly whenever possible.

Sadly, this is not the case in our current implementation. When a request is routed to a peer, and the activity on that peer needs a resource located elsewhere, that peer then has to act like a client and wait for a response, before being able to continue, thus blocking until an answer has been received. One of the reasons for this was, that we were told that we were expected to let our events use the same API as clients, and as the way to use this is blocking, that is what our solution ended up using as well. This might not have been a problem if did not handle requests imperatively, but this was a design constraint we deigned necessary.

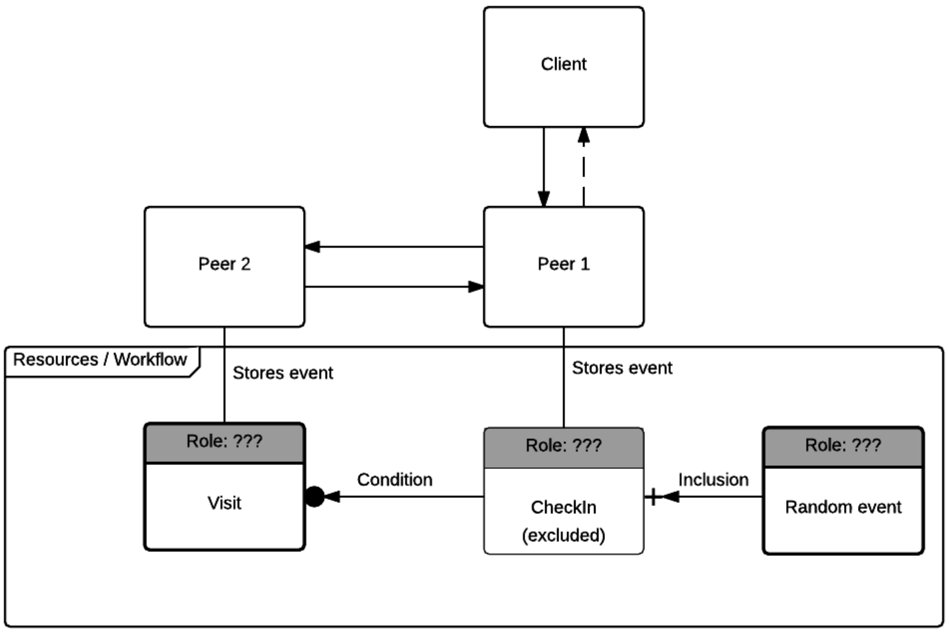


Figure 6.1 – A common deadlock scenario. *The diagram shows two different views: the top one shows the interactions of the peers, and the bottom view that of the DCR-graph activities on these. (Note: The "CheckIn" event is 'excluded' here.)*

CHAPTER 7

Reflection and Conclusion

7.1 Reflection on the project

Programming environment

We decided already from the beginning of the project, that we would attempt to write our entire program using the functional programming language F#, which we had previously been introduced to in the course "Functional Programming" earlier in the semester. The reasons for this were mainly that we were interested in using this language for more than just practice exercises, and additionally had been told by the TAs that it would be a good opportunity to practice it, since we would use it very much on the next semesters.

This choice both had its pros and cons, the main of which are listed here.

Pros:

- We got more experienced in using this language (F#)
- We got a better practical understanding of functional programming
- We got absolutely no errors due to unexpected mutations
- Matches
- Result types are infinitely better than exceptions
- The code itself is likely much shorter than equivalent code written in C# or Java

Cons:

- We spent more time writing things, since we were learning and discovering more

- Some members had more difficulty with the syntax and idiomatic usage
- Automatic type inference errors were sometimes painful to understand
- Sometimes less verbose code is harder to understand

In hindsight, it would probably have been a good idea to have spent more time in the beginning of the project on everybody learning this language, so that we wouldn't have had as many problems with the actual syntax and idiomatic ways of using it, once the actual coding began.

SCRUM

We had some problems with our project management in the first part of the project, due to the fact we had a hard time understanding the given requirements. Due to this, we found it quite hard to plan it (since we wasn't sure what tasks to even plan), so we ended up programming this part of the project ad hoc, sometimes using a bit of pair programming.

For the second part of the project, we had a better understanding of the requirements and the product to make, and this made it more feasible for us to use more planning tools. We decided to use some of the tools of the SCRUM development method, since they would make it easier to get everybody more activated, and make it easier for people to keep an overview of what to work on.

Because it's an academic project and not a 'real' job, the available work hours for the team members varied greatly. Due to this, wrote down how many hours we can work each day for each SCRUM 'sprint'. This had the additional benefit of making it possible for the other group members to see who might show up for group work for any given day, and also who they would be able contact if, in case of difficulties with the work. *See available hours at figure: 8.3 page: 58*

We used a product backlog containing 3 main categories: Code, Report and SCRUM. These were color-coded for easy recognition, and each category would then contain all smaller tasks relating to it. For instance, the 'Report' category contained all of the sections that had be covered in it, such as 'Requirements' and 'Architecture' as tasks.

Each task was then valued with a "priority", where the task with the smallest number should be handled first, usually due to it being necessary to complete, before other tasks could begin. This was mainly the case for the coding, where some parts would logically require the completion of others.

Tasks also had an "Estimate of value", which is a value from 1 to 5, indicating how much we believed that the task would affect the grade and/or overall quality of the product, where 5 indicates the biggest influence and 1, the lowest.

For each task, a member would also volunteer to complete it, and write their name in a spot, so that the other members could see whether a task was in progress or not, and know who to go to in case of delays or such.

The last thing we had for each task, was an "Initial estimate of effort". This value, in hours, indicated how much the work the member currently responsible for the task estimated remained of the task at the end of each day of the sprint. This value was then taken up and reconsidered at the end of each sprint if it hadn't reached completion (0 hours remaining) by then. *See Project backlog at figure: 8.4 page: 59*

In each Sprint meeting, we moved things from the product backlog into a sprint backlog, where people then volunteer to do different tasks and estimate the effort required for them, until each member has tasks equal to the available hours they have indicated for that sprint. *See Sprint 1 backlog at figure: 8.5 page: 60*

7.2 Conclusion

The final product supports functionality to meet all the requirements in sec 1.1, but the support of *role based access control* is very limited as stated in sec 5.1. There are not really any extra features as we were forced to cut all extra functionality that wasn't trivial to implement due to our lack of time.

known Bugs:

1. If we add a peer to the network while its neighbors (the ones who have to migrate content) are very busy, they will probably deadlock. We're not yet sure what causes this.

2. (not technically a bug, but a serious malfeature): Whenever a resource on one peer needs a resource on another, the peer needs to request that resource as a client, thus locking it until it gets an answer. This can in effect make everything deadlock.

3. We’re not certain if you can always execute multiple events at the same time from multiple clients, as we haven’t tested this thoroughly.

4. You can break the data’s consistency and make it difficult to find, by not removing it in the right way.

5. It is not possible to add a peer to the system if the known peer is occupied with something else.

6. There are surely numerous deadlocks possible in our system, but as we haven’t tested very thoroughfully they just haven’t surfaced yet.

7. WorkflowLogic uses EventLogic when deleting events, instead of using the Send module, so it can only delete events on its own peer and an error will occur if the event is not on the same peer as the workflow it is bound to.

Bibliography

[1] Tim Kindberg George Coulouris, Jean Dollimore and Gordon Blair. Webservices. In DISTRIBUTED SYSTEMS Concepts and Design Fifth Edition, page 386. Pearson, 2011. ISBN 9780132143011. URL `http://www.amazon.com/Distributed-Systems-Concepts-Design-Edition/dp/0132143011`.

[2] Wikipedia. Logic gate — wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/w/index.php?title=Logic_gate&oldid=658354839`. [Online; accessed 15-May-2015].

[3] Thomas Hildebrandt Søren Debois and Tijs Slaats. Concurrency & asynchrony in declarative workflows. pages 3–6, 2015.

[4] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Middleware 2001, pages 329–350. Springer, 2001.

[5] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. Distributed Systems: Concepts and Design (5th Edition). Addison Wesley, 5 edition, May 2011. ISBN 0132143011. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0132143011`.

CHAPTER 8

Appendices

dead page

Elaboration of workflow requirements

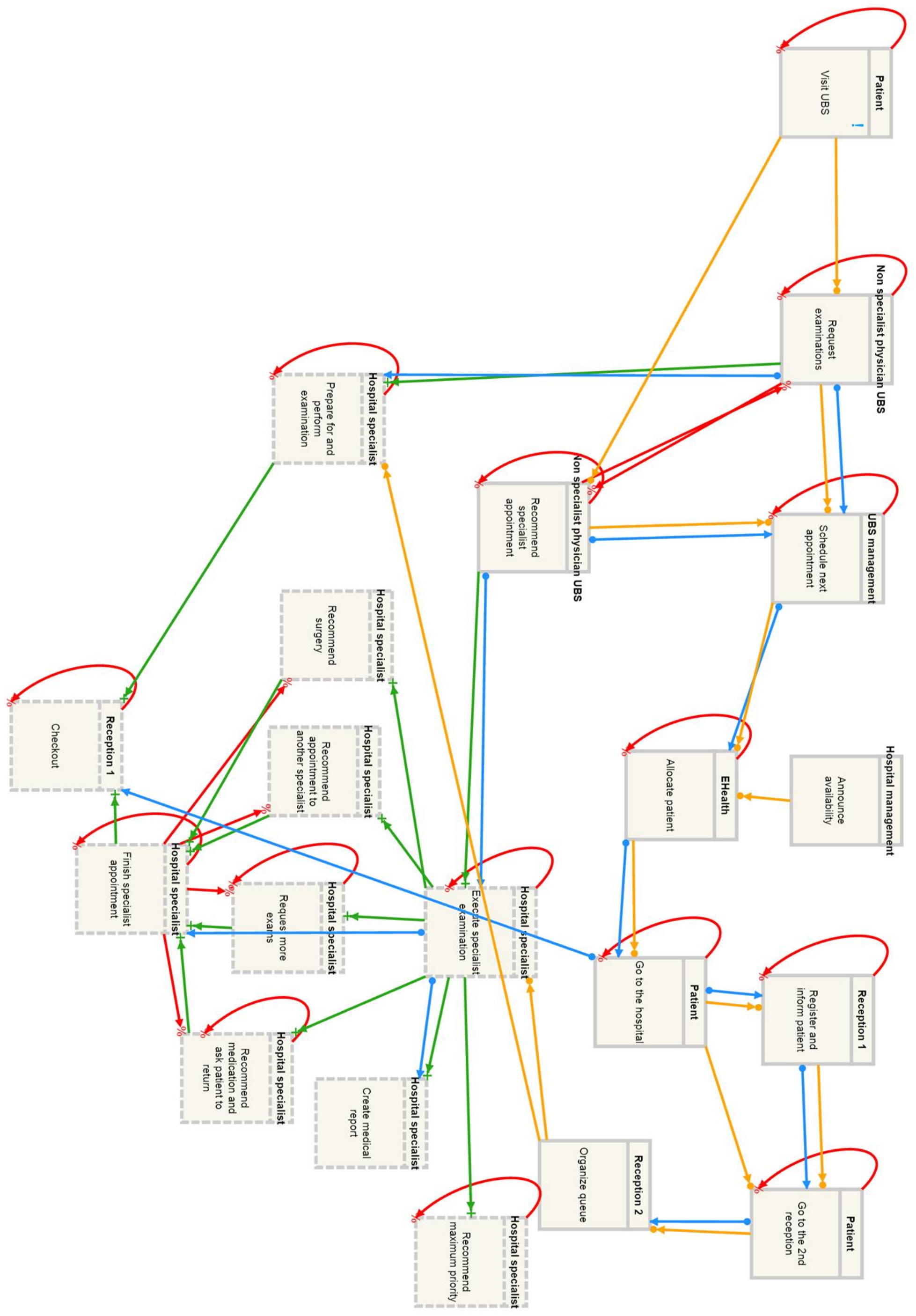


Figure 8.1 – The first version of the hospital DCR-graph.

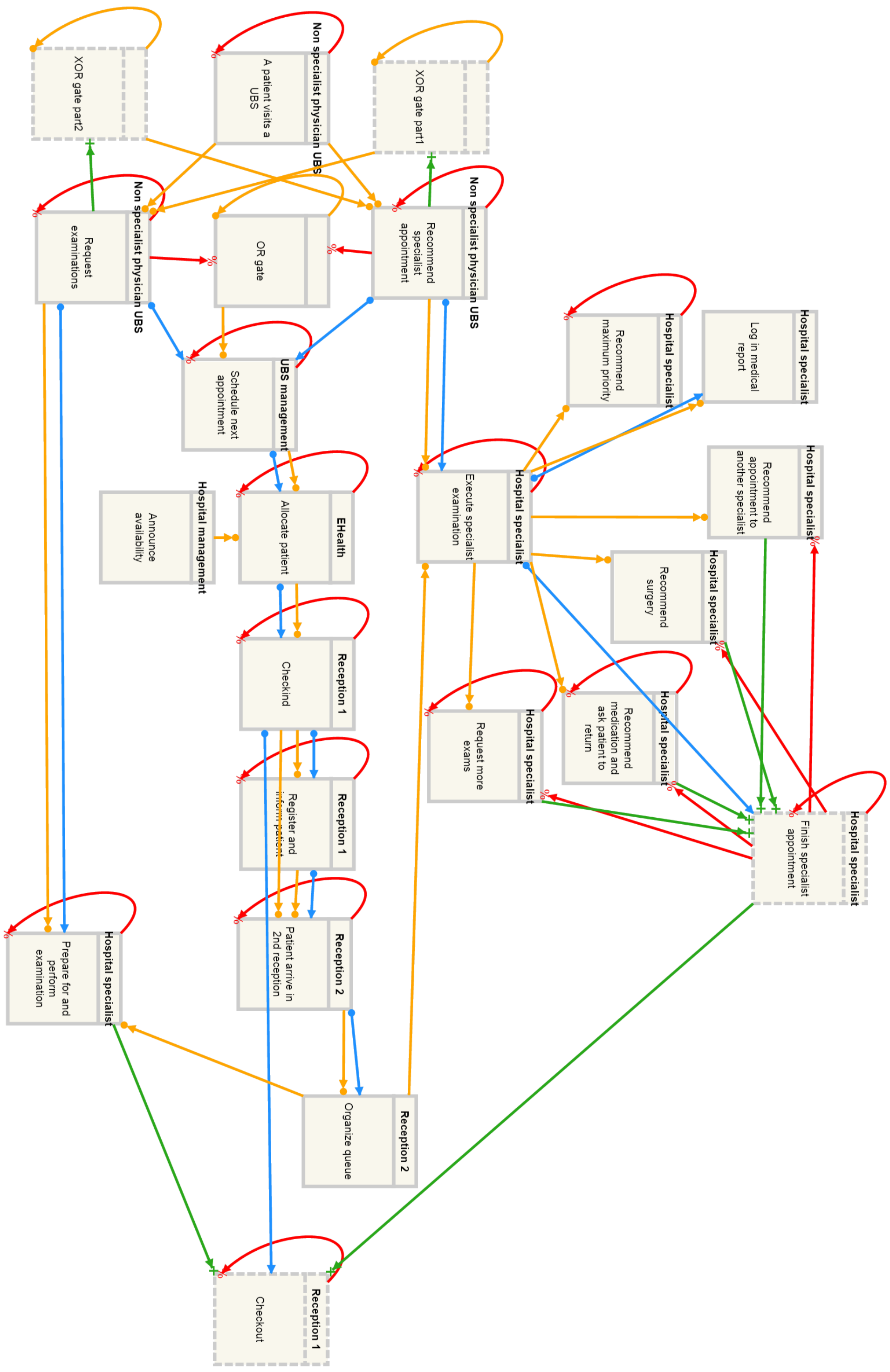


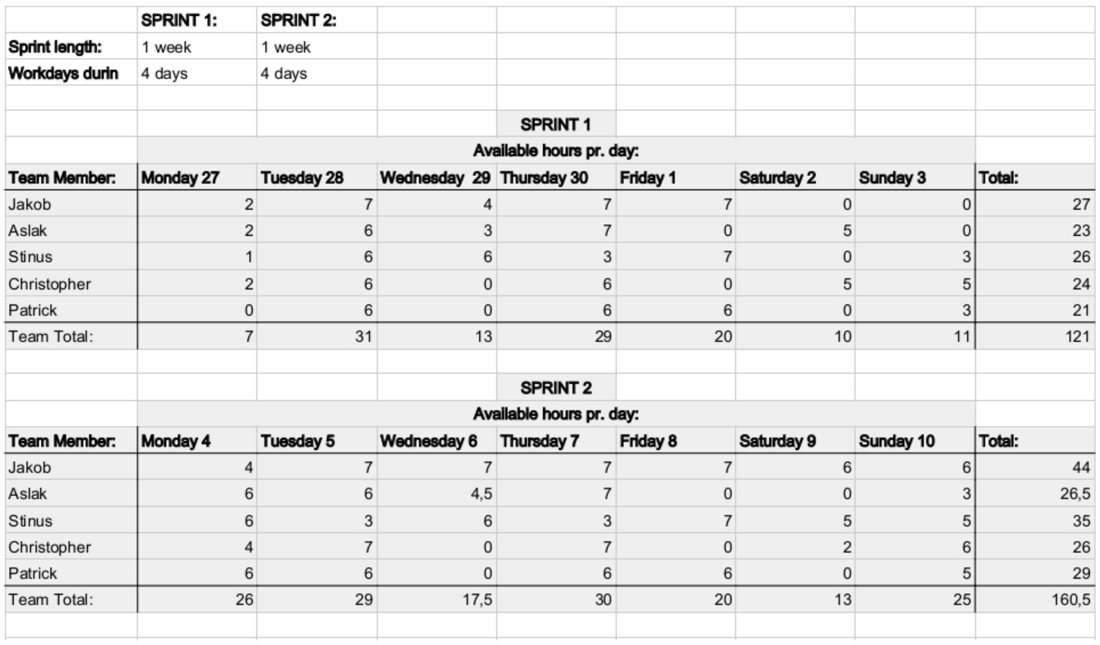
Figure 8.2 – The final version of the hospital DCR-graph, after feedback from Eduardo.

Testing

```
module EventTest.``Testing of the event code``

open NUnit.Framework
open System.IO

let mutable p = new System.Diagnostics.Process()

let download eventPath =
   use w = new System.Net.WebClient ()
   w.DownloadString(sprintf "http://localhost:8080/%s" eventPath)

let testHelpGET eventPath expected =
   let s = download (sprintf "Test/%s" eventPath)
   System.Console.WriteLine("/{0} --> {1}, Expected: {2}", eventPath, s,
       expected.ToString())
   Assert.AreEqual(expected.ToString(),s)

let testHelpPUT eventPath role =
   use w = new System.Net.WebClient ()
   try
      ignore <| w.UploadString(sprintf "http://localhost:8080/Test/%s"
          eventPath, "PUT" , role)
   with
      | x -> System.Console.WriteLine("Exeption: {0}", x.Message)
            0 |> ignore

[<TestFixture>]
type public Test() =

   [<SetUp>]
   member public x.``run before each test``() =
      #if TARGET_MAC
      #else
      if File.Exists("Event.exe")
      then File.Delete("Event.exe")

      let src = @"..\..\..\Event\bin\Debug\Event.exe"
      let dst = "Event.exe"
      File.Copy(src, dst)

      //Starts the server form the .exe fil server from same placement as the
          program.
      p.StartInfo.FileName <- "Event.exe"
      p.StartInfo.Arguments <- ("Test 8080 ")
      p.Start() |> ignore
      #endif

      //System.Console.WriteLine("Start: {0}", startData)
      //System.Console.WriteLine("P: {0} HasExited {1}", p, p.HasExited)

      use w = new System.Net.WebClient ()

      // RESET!
      let testUpload (client: System.Net.WebClient) (url: string) (meth:
          string) (data: string) =
         System.Threading.Thread.Sleep(15);
         client.UploadString(url, meth, data) |> ignore

      testUpload w "http://localhost:8080/Test?action=reset" "PUT" "TestClient"
      testUpload w "http://localhost:8080/Test/Event1" "POST" "000 TestClient"
      testUpload w "http://localhost:8080/Test/Event2" "POST" "000 TestClient"
      testUpload w "http://localhost:8080/Test/Event3" "POST" "000 TestClient"
      testUpload w "http://localhost:8080/Test/Event4" "POST" "000 TestClient"
      testUpload w "http://localhost:8080/Test/Event5" "POST" "000 TestClient"

      testUpload w "http://localhost:8080/Test/Event1/exclusion" "POST"
          "Event2" // [1]->%[2]
      testUpload w "http://localhost:8080/Test/Event1/condition" "POST"
          "Event3" // [1]->o[3]
      testUpload w "http://localhost:8080/Test/Event1/response" "POST"
          "Event5" // [1]o->[5]
      testUpload w "http://localhost:8080/Test/Event2/condition" "POST"
          "Event3" // [2]->o[3]
      testUpload w "http://localhost:8080/Test/Event4/inclusion" "POST"
          "Event2" // [4]->+[2]

      // [1]->%[2], [1]->o[3], [1]o->[5]
      // [2]->o[3]
      // [4]->+[2]
      //
      // [2]-----[3]
      //  | \  /
      //  | [1]
      //  |    \
      // [4]   [5]

   [<TearDown>]
   member public x.``run after each test``() =
      #if TARGET_MAC
      #else
      p.Kill() |> ignore
      #endif
      System.Threading.Thread.Sleep(15);

   [<Test>]
   member public x.``Events start with the right settings`` () =
      testHelpGET "Event1/executed" false
      testHelpGET "Event1/included" true
      testHelpGET "Event1/pending" false
      testHelpGET "Event2/executed" false
      testHelpGET "Event2/included" true
      testHelpGET "Event2/pending" false
      testHelpGET "Event3/executed" false
      testHelpGET "Event3/included" true
      testHelpGET "Event3/pending" false
      testHelpGET "Event4/executed" false
      testHelpGET "Event4/included" true
      testHelpGET "Event4/pending" false
      testHelpGET "Event5/executed" false
      testHelpGET "Event5/included" true
      testHelpGET "Event5/pending" false

   [<Test>]
   member public x.``An event can be executed`` () =
      testHelpGET "Event1/executed" false
      testHelpPUT "Event1/executed" "TestClient"
      testHelpGET "Event1/executed" true

   [<Test>]
   member public x.``Exclusion inhibits execution`` () =
      testHelpGET "Event2/executed" false
      testHelpPUT "Event1/executed" "TestClient"
      testHelpPUT "Event2/executed" "TestClient"
      testHelpGET "Event2/executed" false

   [<Test>]
   member public x.``Condition inhibits execution?`` () =
      testHelpGET "Event3/executed" false
      testHelpPUT "Event3/executed" "TestClient"
      testHelpGET "Event3/executed" false
      testHelpPUT "Event1/executed" "TestClient"
      testHelpGET "Event3/executed" false
      testHelpPUT "Event3/executed" "TestClient"
      testHelpGET "Event3/executed" true

   [<Test>]
   member public x.``A response relation is set to the response value when an
       event is executed`` () =
      testHelpGET "Event5/pending" false
      testHelpPUT "Event1/executed" "TestClient"
      testHelpGET "Event5/pending" true
      testHelpPUT "Event5/executed" "TestClient"
      testHelpGET "Event5/pending" false

   [<Test>]
   member public x.``A response relation is no more set to the response value
       even after the receving event has been executed`` () =
      testHelpGET "Event5/pending" false
      testHelpPUT "Event5/executed" "TestClient"
      testHelpPUT "Event1/executed" "TestClient"
      testHelpGET "Event5/pending" true
      testHelpPUT "Event5/executed" "TestClient"
      testHelpGET "Event5/pending" false

   [<Test>]
   member public x.``An event cannot be pending after it has been executed``
       () =
      testHelpGET "Event5/pending" false
      testHelpPUT "Event5/executed" "TestClient"
      testHelpGET "Event5/pending" false
      testHelpGET "Event5/executed" true
      testHelpPUT "Event1/executed" "TestClient"
      testHelpGET "Event5/pending" true

   [<Test>]
   member public x.``An include relation includes the event`` () =
      testHelpGET "Event2/included" true
      testHelpPUT "Event1/executed" "TestClient"
      testHelpGET "Event2/included" false
      testHelpPUT "Event4/executed" "TestClient"
      testHelpGET "Event2/included" true

   [<Test>]
   member public x.``Inclusion, exclusion and conditions work at the same
       time`` () =
      testHelpGET "Event3/executed" false
      testHelpGET "Event2/included" true
      testHelpPUT "Event3/executed" "TestClient"
      testHelpPUT "Event1/executed" "TestClient"
      testHelpGET "Event3/executed" false
      testHelpGET "Event2/included" false
      testHelpPUT "Event4/executed" "TestClient"
```

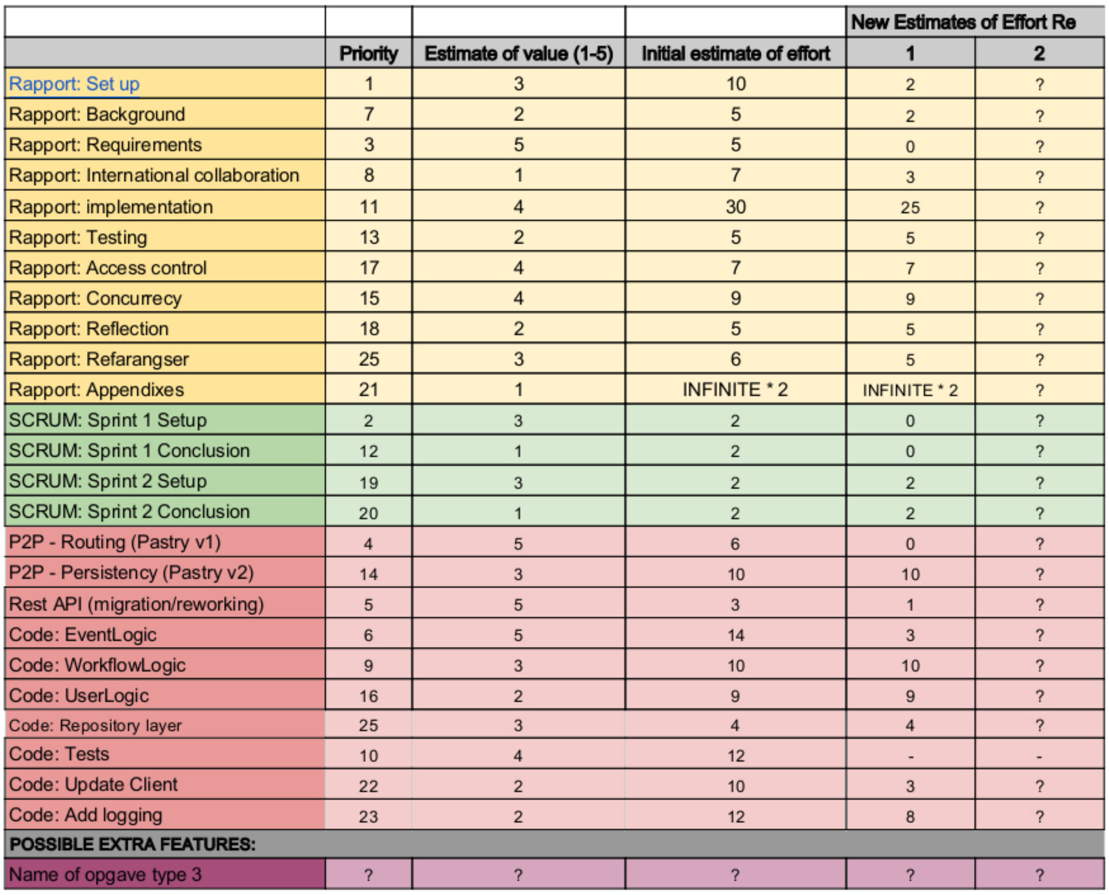
```
      testHelpPUT "Event3/executed" "TestClient"
      testHelpGET "Event3/executed" false
      testHelpGET "Event2/included" true
      testHelpPUT "Event2/executed" "TestClient"
      testHelpPUT "Event3/executed" "TestClient"
      testHelpGET "Event3/executed" true
      testHelpGET "Event2/executed" true

   [<Test>]
   member public x.``An event cannot be executed without the correct role`` ()
       =
      testHelpGET "Event1/executed" false
      testHelpPUT "Event1/executed" "WrongTestClient"
      testHelpGET "Event1/executed" false

   [<Test>]
   member public x.``Get with empty role doesn't fail`` () =
      download "Test?role=" |> ignore

   [<Test>]
   [<ExpectedException( "System.Net.WebException" )>]
   member public x.``Get with unknown action returns 404`` () =
      download "Test?action=pinkfluffyunicorn" |> ignore

   [<Test>]
   [<ExpectedException( "System.Net.WebException" )>]
   member public x.``Get with empty action returns 404`` () =
      download "Test?action=" |> ignore

   [<Test>]
   member public x.``Reset with role does not return `` () =
      let resp = download "Test?role=TEST&action=reset"
      Assert.AreEqual(resp, "Resetting...")
```

Reflection on project and conclusion



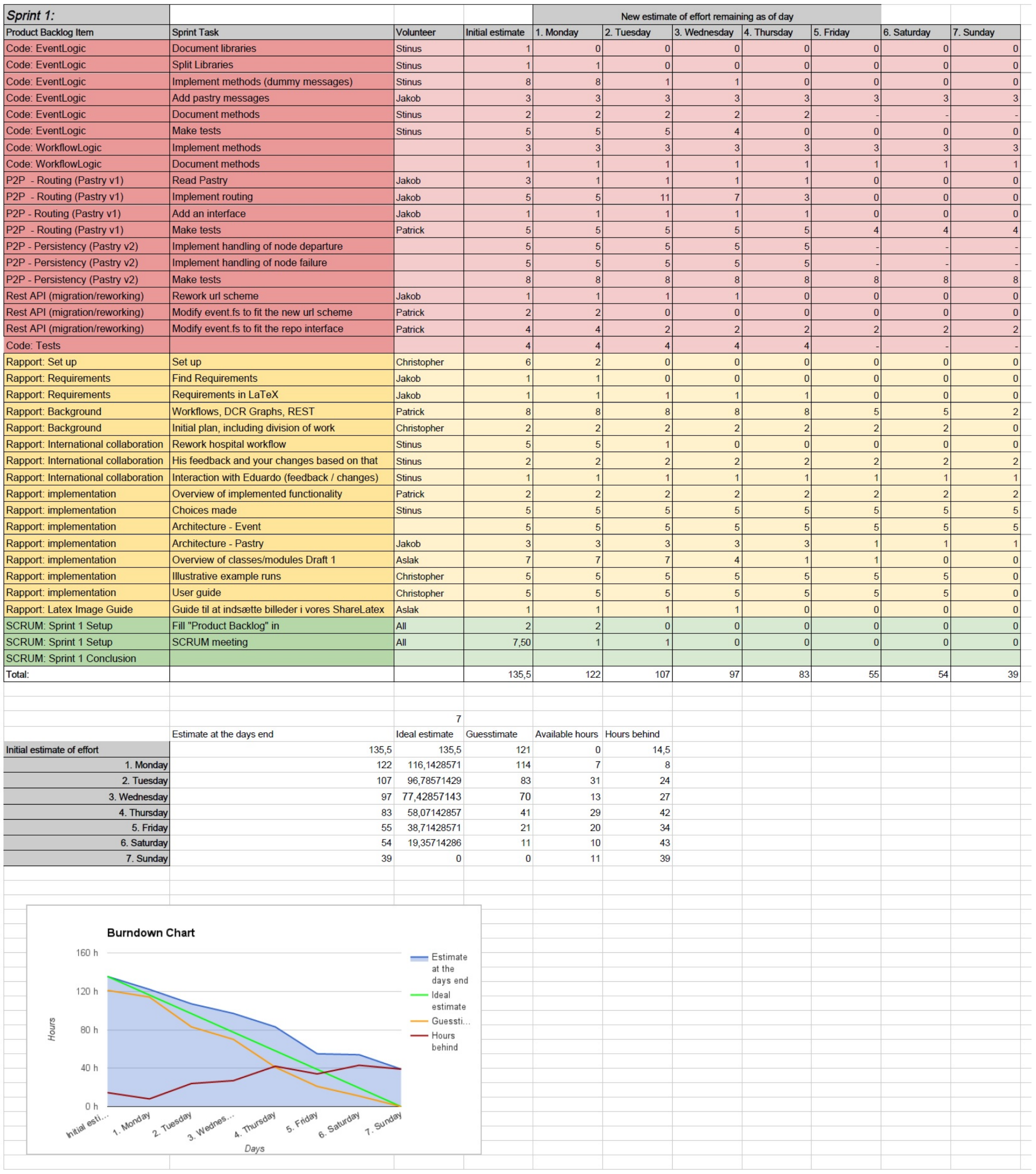
	SPRINT 1:	**SPRINT 2:**						
Sprint length:	1 week	1 week						
Workdays durin	4 days	4 days						
				SPRINT 1				
				Available hours pr. day:				
Team Member:	**Monday 27**	**Tuesday 28**	**Wednesday 29**	**Thursday 30**	**Friday 1**	**Saturday 2**	**Sunday 3**	**Total:**
Jakob	2	7	4	7	7	0	0	27
Aslak	2	6	3	7	0	5	0	23
Stinus	1	6	6	3	7	0	3	26
Christopher	2	6	0	6	0	5	5	24
Patrick	0	6	0	6	6	0	3	21
Team Total:	7	31	13	29	20	10	11	121
				SPRINT 2				
				Available hours pr. day:				
Team Member:	**Monday 4**	**Tuesday 5**	**Wednesday 6**	**Thursday 7**	**Friday 8**	**Saturday 9**	**Sunday 10**	**Total:**
Jakob	4	7	7	7	7	6	6	44
Aslak	6	6	4,5	7	0	0	3	26,5
Stinus	6	3	6	3	7	5	5	35
Christopher	4	7	0	7	0	2	6	26
Patrick	6	6	0	6	6	0	5	29
Team Total:	26	29	17,5	30	20	13	25	160,5

Figure 8.3 – Avalible hours



				New Estimates of Effort Re	
	Priority	Estimate of value (1-5)	Initial estimate of effort	1	2
Rapport: Set up	1	3	10	2	?
Rapport: Background	7	2	5	2	?
Rapport: Requirements	3	5	5	0	?
Rapport: International collaboration	8	1	7	3	?
Rapport: implementation	11	4	30	25	?
Rapport: Testing	13	2	5	5	?
Rapport: Access control	17	4	7	7	?
Rapport: Concurrecy	15	4	9	9	?
Rapport: Reflection	18	2	5	5	?
Rapport: Refarangser	25	3	6	5	?
Rapport: Appendixes	21	1	INFINITE * 2	INFINITE * 2	?
SCRUM: Sprint 1 Setup	2	3	2	0	?
SCRUM: Sprint 1 Conclusion	12	1	2	0	?
SCRUM: Sprint 2 Setup	19	3	2	2	?
SCRUM: Sprint 2 Conclusion	20	1	2	2	?
P2P - Routing (Pastry v1)	4	5	6	0	?
P2P - Persistency (Pastry v2)	14	3	10	10	?
Rest API (migration/reworking)	5	5	3	1	?
Code: EventLogic	6	5	14	3	?
Code: WorkflowLogic	9	3	10	10	?
Code: UserLogic	16	2	9	9	?
Code: Repository layer	25	3	4	4	?
Code: Tests	10	4	12	-	-
Code: Update Client	22	2	10	3	?
Code: Add logging	23	2	12	8	?
POSSIBLE EXTRA FEATURES:					
Name of opgave type 3	?	?	?	?	?

Figure 8.4 – Product Backlog



Sprint 1:				New estimate of effort remaining as of day						
Product Backlog Item	Sprint Task	Volunteer	Initial estimate	1. Monday	2. Tuesday	3. Wednesday	4. Thursday	5. Friday	6. Saturday	7. Sunday
Code: EventLogic	Document libraries	Stinus	1	0	0	0	0	0	0	0
Code: EventLogic	Split Libraries	Stinus	1	1	0	0	0	0	0	0
Code: EventLogic	Implement methods (dummy messages)	Stinus	8	8	1	1	0	0	0	0
Code: EventLogic	Add pastry messages	Jakob	3	3	3	3	3	3	3	3
Code: EventLogic	Document methods	Stinus	2	2	2	2	2	-	-	-
Code: EventLogic	Make tests	Stinus	5	5	5	4	0	0	0	0
Code: WorkflowLogic	Implement methods		3	3	3	3	3	3	3	3
Code: WorkflowLogic	Document methods		1	1	1	1	1	1	1	1
P2P - Routing (Pastry v1)	Read Pastry	Jakob	3	1	1	1	1	0	0	0
P2P - Routing (Pastry v1)	Implement routing	Jakob	5	5	11	7	3	0	0	0
P2P - Routing (Pastry v1)	Add an interface	Jakob	1	1	1	1	1	0	0	0
P2P - Routing (Pastry v1)	Make tests	Patrick	5	5	5	5	5	4	4	4
P2P - Persistency (Pastry v2)	Implement handling of node departure		5	5	5	5	5	-	-	-
P2P - Persistency (Pastry v2)	Implement handling of node failure		5	5	5	5	5	-	-	-
P2P - Persistency (Pastry v2)	Make tests		8	8	8	8	8	8	8	8
Rest API (migration/reworking)	Rework url scheme	Jakob	1	1	1	1	0	0	0	0
Rest API (migration/reworking)	Modify event.fs to fit the new url scheme	Patrick	2	2	0	0	0	0	0	0
Rest API (migration/reworking)	Modify event.fs to fit the repo interface	Patrick	4	4	2	2	2	2	2	2
Code: Tests			4	4	4	4	4	-	-	-
Rapport: Set up	Set up	Christopher	6	2	0	0	0	0	0	0
Rapport: Requirements	Find Requirements	Jakob	1	1	0	0	0	0	0	0
Rapport: Requirements	Requirements in LaTeX	Jakob	1	1	1	1	1	0	0	0
Rapport: Background	Workflows, DCR Graphs, REST	Patrick	8	8	8	8	8	5	5	2
Rapport: Background	Initial plan, including division of work	Christopher	2	2	2	2	2	2	2	0
Rapport: International collaboration	Rework hospital workflow	Stinus	5	5	1	0	0	0	0	0
Rapport: International collaboration	His feedback and your changes based on that	Stinus	2	2	2	2	2	2	2	2
Rapport: International collaboration	Interaction with Eduardo (feedback / changes)	Stinus	1	1	1	1	1	1	1	1
Rapport: implementation	Overview of implemented functionality	Patrick	2	2	2	2	2	2	2	2
Rapport: implementation	Choices made	Stinus	5	5	5	5	5	5	5	5
Rapport: implementation	Architecture - Event		5	5	5	5	5	5	5	5
Rapport: implementation	Architecture - Pastry	Jakob	3	3	3	3	3	1	1	1
Rapport: implementation	Overview of classes/modules Draft 1	Aslak	7	7	7	4	1	1	0	0
Rapport: implementation	Illustrative example runs	Christopher	5	5	5	5	5	5	5	0
Rapport: implementation	User guide	Christopher	5	5	5	5	5	5	5	0
Rapport: Latex Image Guide	Guide til at indsætte billeder i vores ShareLatex	Aslak	1	1	1	1	0	0	0	0
SCRUM: Sprint 1 Setup	Fill "Product Backlog" in	All	2	2	0	0	0	0	0	0
SCRUM: Sprint 1 Setup	SCRUM meeting	All	7,50	1	1	0	0	0	0	0
SCRUM: Sprint 1 Conclusion										
Total:			135,5	122	107	97	83	55	54	39

		7			
	Estimate at the days end	Ideal estimate	Guesstimate	Available hours	Hours behind
Initial estimate of effort	135,5	135,5	121	0	14,5
1. Monday	122	116,1428571	114	7	8
2. Tuesday	107	96,78571429	83	31	24
3. Wednesday	97	77,42857143	70	13	27
4. Thursday	83	58,07142857	41	29	42
5. Friday	55	38,71428571	21	20	34
6. Saturday	54	19,35714286	11	10	43
7. Sunday	39	0	0	11	39

Figure 8.5 – Sprint 1 Backlog