

\section*{Contents}

\section{Introduction}

\section{Planning}
\subsection{Introduction}
\subsection{Requirements}
\subsection{Design}
\subsubsection{The land algorithm}
\subsubsection{The Naive algorithm (ships)}
\subsubsection{The CLP algorithm (ships)}
\subsection{Implementation}
\subsection{Discussion}
\subsubsection{The Requirements}
\subsubsection{Proposed extensions}
\subsubsection{Performance}
\subsubsection{Other algorithms}

\section{Process and database management}
\subsection{Introduction}
\subsection{Process management}
\subsubsection{Requirements}
\subsubsection{Design}
\subsubsection{Implementation}
3.3  Database management ........................................ 30
  3.3.1  Requirements ............................................. 31
  3.3.2  Design .................................................... 31
  3.3.3  Implementation .......................................... 32
3.4  System integration ............................................ 36
  3.4.1  Requirements ............................................. 36
  3.4.2  Design .................................................... 37
  3.4.3  Implementation .......................................... 38
3.5  Discussion ................................................... 39
  3.5.1  Optimizations ............................................. 40
  3.5.2  Extensions and Omissions ................................ 40

4  User interface ................................................. 41
  4.1  Introduction ................................................ 41
  4.2  Requirements ................................................. 42
  4.3  Preplanning activities ....................................... 43
    4.3.1  Human-computer interface design ...................... 43
    4.3.2  Subsystem design ..................................... 46
    4.3.3  Implementation ....................................... 48
    4.3.4  Human-computer interface implementation .......... 48
  4.4  Postplanning activities ..................................... 49
    4.4.1  Design ................................................. 49
    4.4.2  Implementation ....................................... 50
  4.5  Discussion ................................................... 52
    4.5.1  Object-orientation in Erlang .......................... 52

5  Conclusions .................................................... 55
List of Figures

3.1 Server structure, and the relationships between all clients: the system manager (SM), the drawing tool, and the analysing tool (both named Client in the figure). There can be several instances of the components marked with a star (*), on the same or on different machines. All components in the same square must run on the same machine; components in different squares can, but do not have to, run on different machines. .......................... 22

3.2 The base window of the system manager with four different machines running. The process distribution follows the load — no processes at all are running on mamba.csd.uu.se since it has a high load. The tables for the machines have entries for all main processes running on them. ............... 28

3.3 The node management window for adding and removing nodes dynamically at runtime. ................................................................. 28

3.4 The user management window. User ‘kalle’ is currently logged on. ....... 29

3.5 The resource editor for the host files. ................................. 30

4.1 The drawing tool with a scenario. Stores are indicated by filled circles, quays by filled squares, minings by opaque filled rectangles, and crossings by opaque circles. ............................................................ 44

4.2 The analysis tool, displaying the results of the laying of the minings. The results are represented by DD HH:MM:SS from start. ............... 51
Chapter 1

Introduction

Plan95 is a distributed planning system developed in a student project at the Computing Science Department, Uppsala University. The project involved 20 MSc-level students. General design and early prototyping was done by Björn Carlson. The project was coordinated by Björn Carlson and Håkan Millroth, with assistance of Jan Gabrielsson. Industrial partners were CelsiusTech IT AB and Telub AB.

The pedagogical goals of the project were:

- To train the students in working with others in a (comparatively) large project.
- To demonstrate that with advanced tools and sound methodology, it is possible to design and develop quite complex systems in a short period of time.

From an abstract point of view, the problem solved by Plan95 can be seen as planning the transportation of items from source stores to destination stores with reloading en route. There are two types of vehicles: for transportation from source nodes to reloading stores, and for transportation from reloading stores to destination stores. Each store and vehicle can hold a fixed number of items. Each transportation between two stores has a fixed cost. The problem is to schedule the vehicles so that the total cost for the transportation of all items to the destination stores is minimized.

In our concrete application, items are naval mines, source stores are land stores, reloading stores are quays, destination stores are minings, vehicles are trucks and ships.

The mission of the project was to design and develop an interactive software system for stating, solving and analyzing problems of this type.

Plan95 is a distributed system divided into three subsystems:

- The planning subsystem uses constraint logic programming [5] and is written in PROLOG [3, 4].
- The *process and database management* subsystem is written in *ERLANG* [1].
- The *graphical user interface* is written in *ERLANG* and Tcl/Tk [6].

The subsystems are described in Chapters 2–4.

The time spent by each student on the project was six weeks, corresponding to a total of about 30 man-months. The system consists of 40000 lines of Erlang code, 5500 lines of Prolog code, and 1000 lines of Tcl code.
Chapter 2

Planning

By Cecilia Ekelin, Anders Jackson, Patrik Manlig, Gustaf Naeser, Martin Olofsson and Jan Sjödin.

2.1 Introduction

A *scenario* contains components (mines), resources (trucks and ships) and a map. The map is an undirected graph with nodes of four types: stores, quays, crossings and minings. There are two types of edges: roads and waterways. All nodes have a position, and all edges have a distance. Store nodes have a capacity and can contain resources (only trucks) and components (mines). Quay nodes also have a capacity and can contain resources (both trucks and ships). Minings can have priorities and demands for mines.

The planner is the part of the system that, from a scenario, designs a *plan*, which describes when and where the trucks and ships are loaded and unloaded. There are some constraints on plans that restrict what is allowed and what is not allowed in a plan. The problem is not only distributing the components, but to do so in an order where destinations with higher priorities are supplied earlier than those of lower priority. Obvious constraints are that resources should be used correctly, e.g. ships are only used at sea and not at land. A full description of what a plan is will be given in section Section 2.2.

The planner is a module in the system. The planning is supervised by an Erlang node which handles the communication with the server. The Erlang node starts a Prolog\(^1\) that performs the actual planning.

There are two different algorithms implemented for the planning, one naive algorithm and one algorithm based on Constraint Logic Programming (CLP).

The problem given by a scenario is a search problem. The search space is large and if the problem is to be solved efficiently, this space has to be drastically pruned. The planner

\(^1\)In our case SICStus Prolog v3.0pl3(FD) beta
tries to do this by applying different search strategies.

The constraint programming paradigm lets the programmer specify the problem in terms of constants, variables and expressions. The constraint solver then uses the expressions to constrain the possible values of the variables. A CLP language provides built-in search algorithms for finding assignments to the variables.

2.2 Requirements

The requirements on the planner are mainly functional and mostly describe constraints on the scenario, how a plan may be derived and needed characteristics of a plan. The nonfunctional requirements that were present in the early stages in the project concerned the performance of the planner. As these demands are hard to verify they will be studied in the discussion (Section 2.5).

The requirements in the following presentation are not ordered by importance.

Nodes

- There are four types of nodes: stores, quays, minings and crossings.
  - Stores are located on land and can only be reached by trucks. This is where the components are stored.
  - The quays are connections between land and water transportation. Both ships and trucks can be present in quays.
  - Minings are destinations for mines. All minings are located at sea.
  - Crossings are nodes which just exist to join any number of roads.

- The stores have a capacity that gives the maximal number of mines that can be stored. The capacity may never be exceeded.

- Stores and quays have queues that influence the performance of the node. Lots of queues gives the node a better chance of meeting the services requested from it:
  - In stores the queues are assembly queues and a store with a greater number of queues will be able to assemble mines faster than one with fewer queues.
  - In quays the queues are used for lading and a quay with many queues can have more ships loading mines at the same time.

- Minings are prioritized. Priorities are given as integers where 1 is the highest priority and the succeeding integers are of successively lower priority. The priority of two minings can be the same.
• Minings have a demand of mines. This demand must be fully met or else the mining should not get any mines at all. The implication of this is that if there are not mines enough to fill a mining, a mining of lower priority can be filled if the available mines are sufficient to supply its demand.

Mines

• The system should provide support for different types of mines.

• Mines are initially located in stores.

• There can be mines whose initial position is unspecified. The initial placement of these mines is decided by the planner.

• Mines have an associated assembly time that tells how long time it takes to ready the mine for transportation.

• Mines, of any type, can be regarded as having the same physical size, weight and volume.

Trucks and ships

• There are different models of trucks and ships.

• Trucks are initially located in stores while ships are initially located in quays.

• There can exist trucks and ships whose initial position is unspecified. The initial position of these may be decided by the planner (see Section 2.3.1 for trucks and Sections 2.3.2 and 2.3.3 for ships).

• Each truck and ship has a maximum speed. This speed can never be exceeded but the trucks and ships are free to travel at any speed slower than it.

• All trucks and ships have a capacity which indicates the maximum number of mines that they can carry at any time.

• Ships have rails of a specified length. The number of rails affect the time it takes the ship to drop mines.

• Specified ship types can be disallowed at certain quays.
Plans

- A plan is a schedule for the trucks and ships which meets the above requirements.
- All mines must be present in a quay for a ship to be able to load them.
- The scenario must have storage capacity to store all mines. This is vital when the planner has unplaced mines to find the initial placement for.
- A plan should use as short time as possible.

2.3 Design

Since the most parts of the system is written in Erlang but the planner in Prolog, there is an Erlang interface node which supervises the planning. From the planner’s point of view this node supplies the planner with a scenario and collects the plan when the planning is complete. This section only covers the Prolog part of the planner. The Erlang interface is described in Section 3.4.2.

The goal of the planning is that mines should be transported to and dumped in the mining. Mines will first be moved by trucks from a store to a quay, and then by ships to their designated mining. The planner can be designed to do this in one or two steps. A one-step scheme would plan both the land and water transports in a single pass. A two-step scheme would first plan one of the steps and then the other. If the last step in the chain of transportations, the water transports, can be carried out without any delays, all the steps prior to it can satisfy its demands; the two-step scheme is satisfactory.

A two-step planning will have a harder time finding solutions minimizes truck or ship usage than its one-step equivalent. The planners used in this system are two-step variants.

Before the search for a plan starts, there is one important task to be solved. As the scenario generally does not contain enough mines to fill all the mining, a decision which minings to mine has to be made. With the minings having priorities, this task can be reduced to trying to fill the minings in priority order.

2.3.1 The land algorithm

The land algorithm tries to satisfy the orders generated by the water algorithm (naive or CLP) as fast as it can. The orders are ordered by priority, with the first order from each ship being filled before the second order of any ship can be filled, so that as many ships as possible avoid being idle. It is assumed that this does not slow down the ships significantly, since the trucks are considered a non-critical part of the problem. The trucks pick mines from a store that can supply them in the least amount of time (see section 3.2
for the limitations on nodes) and then transport the mines to a quay in order to fill the
most urgent order.

We introduced two restrictions on the algorithm:

1. Disjoint graphs, or graphs with disjoint sub-graphs, are not handled.

2. The trucks do not try to take a full load if the selected order is not enough to fill the
truck.

2.3.2 The Naive algorithm (ships)

The naive algorithm was created to make it easier to develop the communication between
the modules. The purpose of this approach was to test a full system without needing to
painfully hand code samples. The algorithm handles all correct inputs and produces a
correct, but not necessarily good, plan.

The algorithm produces jobs for the trucks and schedules for the ships. A job contains
mines that should be transported to a specific quay. Each job is associated with one
specific mining. When a ship loads mines at the quay, a job is created. The priority of
a job depends on the priority of its mining. The algorithm plans the routes of the ships
to supply the minings with the highest priority first. If there are several minings with the
same priority, the algorithm would randomly choose any of those minings. However, the
one it does choose is not necessarily the best choice. A single route is planned by selecting
a ship and a mining. The ship moves to the closest quay and loads all the mines for the
selected mining. Then it moves to the mining and unloads. This routine repeats until
there are no more minings to supply.

A ship mines a complete mining at a time. This means that two ships will never share
minings.

Restrictions

1. The demand of every mining must be satisfiable using only one ship.

2. Constraints concerning docking ships are ignored.

2.3.3 The CLP algorithm (ships)

The planning problem is quite complex, so it is rather difficult to develop an algorithm
that can handle the problem accurately. The CLP facility of PROLOG provides built-in
procedures to minimize arithmetic expressions. Thus, we can express the planning problem
as arithmetic expressions, constraints, and then minimize them.
We chose to divide the planning into two parts to simplify the problem. The parts are flow analysis and scheduling. The flow analysis determines the number of mines transported on each waterway. The scheduler then decides which ship is to be used where and when, to carry out the transportations.

**The flow analysis**

The purpose of the flow analysis is to minimize the cost of mines transported on each waterway. The expression to be minimized is the sum of, for all waterways: the sum of all mines transported on the waterway multiplied with the cost (length) of the waterway.

The flow analysis does not consider time in its computation. Furthermore, it uses capacity (as in stores) and not the number of queues of a quay. Therefore, some pre-processing has to be done to convert queues to capacity. The conversion must take into account that since time is not considered, all mines must be able to reside in the quays at the same time.

The constraints needed for the flow analysis are:

**flow constraint 1** The sum of all mines transported to a mining must equal the demand of the mining.

**flow constraint 2** The sum of all mines transported from a quay is less than or equal to the capacity of the quay.

Note that the flow analysis does not consider the priority of a mining or where the ships are placed.

**The scheduler**

The purpose of the scheduler is to minimize the time to mine a priority group, i.e. all minings with the same priority, starting with the group with highest priority. This is done incrementally forcing each new end time to be 5% better than the previous one.

The flow analysis provides the number of mines to be transported from a quay to a mining but since the scheduler deals with priority groups, the output from the flow analysis must be processed to suit the scheduler. The processing constructs jobs which describe from which quays a mining gets its mines.

The scheduler builds an expression consisting of the time it takes for the ships to carry out all jobs with the same priority. The time expression for each job is basically the time for the ships to move from its current position to the quays in the job, load the mines and transport them to the mining. To prevent that a ship is used in several places at the same time, the expression is built under the assumption that a ship only can be used once in a priority group. If there are too few ships to schedule a whole priority group with this assumption, the scheduler only considers a job at a time. Once a group has been scheduled
the scheduler continues with the next group, with the ships that have been used located at their new positions. The same holds if the scheduler considers a job at a time.

The constraints needed to obtain a schedule that meets the requirements are:

**schedule constraint 1** The sum of all mines transported to a mining must equal the amount in the job.

**schedule constraint 2** The number of mines that a ship carries must not be greater than the ship’s capacity.

**schedule constraint 3** The number of ships that load from a quay at any time must not be greater than the number of queues at the quay.

**schedule constraint 4** If several ships have to be used to complete a mining they must start the mining simultaneously.

**schedule constraint 5** A ship that is forbidden in a quay must not be used there.

The additional constraints needed due to the use of priority groups are:

**job constraint** A ship can only be used once in a job.

**group constraint** A ship can only be used once in a priority group.

**Restrictions**

The number of quays determined by the flow analysis, involved in a job must not be greater than the number of available ships.

## 2.4 Implementation

Plans are implemented as event lists. An event consists of an action, a position where the action takes place, the current load of the resource, and a time when the action is completed.

There are five possible actions for an event:

- *initial* indicates where the resource starts the planning
- *load* and *unload* tells that the resource loads or unloads mines
- *arrival* and *departure* is used to inform that the resource arrives at or departs from a node.
- *delay* is used to indicate that the resource has to wait
2.4.1 The land algorithm

Each truck is associated with a ‘time elapsed’ attribute that is the total time to execute the given truck’s current schedule. Initially, the schedule of each truck is empty (and thus the ‘time elapsed’ is zero). The truck with least time elapsed is then chosen to bring one load of mines to a quay in order to fill the most urgent job. If several trucks have the same (least) time elapsed, one of them is chosen. This will not use the trucks in an optimal way. However, it will be good enough for our purposes. The schedule of the truck is then appended with the action of transporting the selected load to the selected quay.

Algorithm description:

1. Find the truck with least time elapsed.
2. Select an action for the truck:
   (a) If there are any mines in the truck’s current location, the algorithm selects an order that can be (partially) filled with these mines. The truck’s next action will be to transport as many mines as possible to fill the selected order. The most urgent order is considered first, but if that cannot be filled, any order will do. The truck will possibly delay while the mines are being readied for transport.
   (b) If there are no mines in the truck’s current location, the algorithm searches for a location that contains mines. The truck’s next action will be to move to the closest such location. The reason for selecting the closest location is to minimize the time that a truck is not carrying any load. If no such location can be found, the algorithm halts (there are no more mines to transport).
3. The schedule of the truck is appended to reflect the chosen action.
4. Repeat until the iteration halts in step 2(b).

2.4.2 The Naive algorithm (ships)

The unplaced ships are randomly placed at the quays. Two extra attributes are added to every ship: a time variable and a job number. The time variable keeps track of a ship’s actual time. This is a constrained variable that will be minimized when the total planning is completed. A ship’s regular time attribute keeps track of the time disregarding all delays due to synchronizations with trucks. A time variable is constrained to be greater than or equal to the previous time variable plus the time it takes to complete the action the ship is currently performing. That time variable should also be greater or equal to the time when the job is finished for that load. We need the extra constraint because, if a job is completed after the ship arrives, the ship must wait before loading.

Algorithm description:
1. Select the ship with the least ‘time’ attribute and select a suitable mining. A suitable mining is the one with the highest priority of all the minings that are small enough for the ship to fill. If there are no suitable minings (possibly none left to mine) the ship is considered to have a complete schedule, and a new ship is selected. If there are no ships left the algorithm terminates.

2. Move the ship to the closest quay if it is not already in a quay. This will add a departure and an arrival event to the ship’s event list.

3. Load the mines for the selected mining and create a job. The job is created so the truck planner will know how many mines that needs to be transported to the quay where the ship is located. This will result in a load event, and a new time variable. The new time variable $T'$ has the constraints $T' \geq T + L + MD$ and $T' \geq J + L$, where $T$ is the old time variable, $MD$ is the time it takes to move to the quay (0 if the ship is already at a quay), $J$ is the time when the job is completed, and $L$ is the time it takes to load the mines.

4. Move the ship to the chosen mining. This will also add departure and arrival events to the ship’s event list.

5. Unload the mines. The time attribute is updated with the time added for the planned route and a new time variable is used. The new time variable $T''$ has the constraint $T'' \geq T' + U + MM$, where $MM$ is the time it takes to move to the mine field, and $U$ is the time to unload the mines. An unload event is added to the ship’s event list.

2.4.3 The CLP algorithm (ships)

To use CLP, the problem has to be declared in expressions. The expressions consist of variables and constants. The variables that describe properties of the problem. The issue is to assign values to the variables so that the expression is minimized and yet the constraints are not violated. The flow analysis is implemented in CLP(Q) which provides a minimize algorithm. The scheduler is implemented in CLP(FD) which can handle nonlinear expressions and also allows gradual improvement of the schedule.

The flow analysis

Before the actual flow analysis we have to do the conversion from queues to capacity. This is done in priority order to ensure that the highest prioritized minings get the best service. The algorithm for computing the capacity is:

1. Distribute the number of mines in the mining into quays that can be reached from the mining. The amount given to a quay is reflected by the number of queues in the quay.
2. Repeat step 1 for all minings.

3. Add the amounts of mines for a quay together and let this be the capacity of the quay.

A small number of queues in a quay means that fewer number of ships can load at the same time. This can cause delays if too many mines are to be transported via the same quay. The algorithm reflects this property by assigning a smaller (larger) capacity to those quays with few (many) queues and thus not allowing too much to be transported via a quay with few queues. However, despite that the flow analysis does not care about time, quite a lot can be transported via a quay with few queues as long as it is not done at the same time. Therefore an overhead-capacity is added to each quay. The overhead-capacity is computed in a way similar to how the mines are distributed over the quays.

**Variables.** Since CLP(Q) is used, the variables’ values are in the domain of rational numbers.

If $t$ is the number of mine types and $e$ the number of edges, there are $t \times e$ variables $T_{i,j}$ representing the number of mines of type $i$ that are transported on the edge $j$.

The constraints described in the design are expressed with the variables as:

**flow constraint 1** Let $demand(i)$ be the demand of mine type $i$ in a mining. In addition, let $(m,q_1),\ldots,(m,q_n)$ be the edges between mining $m$ and quays $q_1,\ldots,q_n$. Then for each mining and each mine type $t$ the following should hold:

$$demand(t) = T_{t,(m,q_1)} + \cdots + T_{t,(m,q_n)}$$

**flow constraint 2** Let $(q,m_1),\ldots,(q,m_n)$ be the edges from a quay $q$ to minings $m_1,\ldots,m_n$ and let $t_1,\ldots,t_m$ be the mine types and $capacity(q)$ be the capacity of a quay $q$. Then for each quay the following should hold:

$$capacity(q) \geq (T_{t_1,(q,m_1)} + \cdots + T_{t_1,(q,m_n)}) + \cdots + (T_{t_m,(q,m_1)} + \cdots + T_{t_m,(q,m_n)})$$

Note that this is because time is not considered during the flow analysis.

Let $e_1,\ldots,e_n$ be all the edges, $t_1,\ldots,t_m$ be the mine types and $cost(i)$ be the length of edge $i$. Then we should minimize:

$$(cost(e_1) \times T_{t_1,e_1} + \cdots + cost(e_n) \times T_{t_1,e_n}) + \cdots + (cost(e_1) \times T_{t_m,e_1} + \cdots + cost(e_n) \times T_{t_m,e_n})$$
The scheduler

The scheduler considers jobs which are to be scheduled. A job represents:

- The priority of the mining.
- In which quays the mines will be located, when and how many.

The jobs are constructed from the output of the flow analysis. However, since we need to know when the trucks have delivered the mines in each quay the truck planning is done before the ship scheduling.

Variables. Since CLP(FD) is used, the variables’ values are natural numbers (including zero).

The variables used to express the problem of scheduling a job is for each ship:

- Let $q$ be the number of quays in the job, then there are $q$ variables $B_{i,j}$ in the domain $0..1$ where $B_{i,j}$ represents whether the ship $j$ uses quay $i$ or not.
- Let $t$ be the number of mine types and capacity be the ship’s capacity, then there are $t$ variables $L_{i,j}$ in the domain $0...capacity$ where $L_{i,j}$ represents how much the ship $j$ loads of mine type $i$.
- Let $max$ be the sum of all ships loading times, then there is a variable $D_j$ in the domain $0...max$ that represents how long the ship $j$ will be delayed by other ships loading at the same quay. Such delay occur when there are more ships than queues in the quay.

Let $to\text{quay}(i, j)$ be the time for a ship $i$ to travel from its current position to $j$, $to\text{mining}(i, j)$ be the time for a ship $i$ to travel from quay $j$ to the mining, $load(i)$ be the time for the ship $i$ to load the mines expressed in the $L$ variables, $available(j)$ be when the mines are available in the quay $j$ and $initial(i)$ be when the ship $i$ is available. Then

$$start\text{loading}(i) = max(available(j), initial(i) + to\text{quay}(i, j)) + D_j$$

$$departure(i) = start\text{loading}(i) + load(i)$$

The constraints described in the design (Section 2.3.3) are expressed with the ship variables as:
schedule constraint 1 Let $amount(i,j)$ be the amount of the mine type $j$ in quay $i$ and $s_1, \ldots, s_n$ be the ships. Then for each quay $q$ and for each mine type $t$ the following should hold:

$$amount(q,t) = B_{q,s_1} \cdot L_{t,s_1} + \cdots + B_{q,s_n} \cdot L_{t,s_n}$$

schedule constraint 2 Let $capacity(i)$ be the capacity of ship $i$ and $t_1, \ldots, t_m$ be the mine types. Then for each ship $s$ the following should hold:

$$capacity(s) \geq L_{t_1,s} + \cdots + L_{t_m,s}$$

schedule constraint 3 To satisfy this constraint, ships used in all previous jobs must be considered. We define $overlap(x_1,x_2,y_1,y_2)$ to be 1 iff $x_1 \geq y_1$ and $x_1 \leq y_2$.

Let $s_1, \ldots, s_n$ be all the ships at a quay (both previous and current), $U_j$, $V_j$ be $start\ loading(j)$, $departure(j)$ of ship $j$ at a quay and queues the number of queues at the quay. Then for each quay $q$ and ship $s$ the following should hold:

$$queues \geq B_{q,s} \cdot B_{q,s} \cdot overlap(U_s, V_s, U_{s_1}, V_{s_1}) + \cdots + B_{q,s} \cdot B_{q,s} \cdot overlap(U_s, V_s, U_{s_n}, V_{s_n})$$

schedule constraint 4 To satisfy this constraint let all ships in a job wait at the mining until all ships assigned the job have made it to the rendez vous.

schedule constraint 5 Let $s$ be a ship that is forbidden in quay $q$. Then the following should hold:

$$B_{q,s} = 0$$

job constraint Let $q_1, \ldots, q_n$ be all quays in a job. Then for each ship $s$ the following should hold:

$$1 \geq B_{q_1,s} + \cdots + B_{q_n,s}$$

group constraint Let $J_{i,j}$ be the sum obtained in the job constraint for a ship $i$ in job $j$ and $j_1, \ldots, j_m$ be the jobs in the priority group. Then for each ship $s$ the following should hold:

$$1 \geq J_{s,j_1} + \cdots + J_{s,j_m}$$
Then we need an expression for the time to carry out a job which should be minimized. The time for a ship \( s \) to use a quay \( q \) in a job is:

\[
B_{q,s} * (\max(\text{available}(q), \text{initial}(s) + \text{to\_quay}(s,q)) + D_s + \text{load}(s) + \text{to\_mining}(s,q))
\]

The time for the job is then the maximum of the times for each ship and for each quay. Note that if a ship is unplaced, \( \text{to\_quay}(s,q) \) is zero which means that if the ship is chosen the ship is initially placed in \( q \).

To achieve a better schedule the scheduler first tries to consider a whole priority group before it determines the schedule. To achieve that, the scheduler iterates the constraining process for all jobs in the priority group.

The time to minimize for a priority group is the maximum of all the job times.

Once the constraints for a priority group have been set up, the \( B \), \( D \) and \( L \) variables are assigned values that do not violate any constraints. If this is not possible for a whole priority group, the constraints are set up and the variables assigned values for each job instead. After each instantiation the ships are given their new events and position. The resulting times are used in the next run of the scheduler to impose the extra constraint that the next job time should be to 5\% smaller.

### 2.5 Discussion

The planner implemented is a fairly good one. The complexity of the problem comes mainly from the fact that the solution space is enormous. The planning must not bog down the myriad of symmetries present, e.g. two identical trucks can do each others’ jobs just as good. The main problem is therefore to cut the solution space as much as possible but while doing this much of the generality of the planner is sacrificed. In the rest of this section different aspects of these sacrifices among with proposed expansions will be discussed.

#### 2.5.1 The Requirements

During the development some of the requirements were found to be unnecessarily strict or constraining. Ideas of how they could be modified arose and the more interesting ones will be presented here with a short description of their expected impact on the planner.

**Initial positions of the objects.** If the ships where able to start ‘at sea’, it would be easier to simulate more accurate scenarios since ships most often do not lay anchored waiting for war to break out. Another quite severe restriction in the current system is that the mines must be in the stores at the start of the planning. It might be more realistic if
mines could be on trucks or even on ships. The relaxation of these restrictions would give a more flexible planner.

**Smarter queues.** The queues in quays are not as intelligently used as one would hope. In the current CLP implementation they are used only to determine how many ships that should load at the same time. One might instead use them as a speed measure in which ships could use more than one queue to reduce the loading time.

**The capacity notion.** By giving all mines the same physical size, weight and volume the complexity of the problem is pleasantly reduced. This, however, removes some very important attributes of the real world problem.

**Loading mines.** The fact that all mines should be present in the quay for a ship to be able to start loading forces the planner to wait unnecessarily long. Since the mines are dropped in a specified order into the mining, they must be loaded in the reverse order in which they are to be dropped. This impede the planner to optimize the loading time, it cannot start even if some of the mines could be loaded. To relax this restriction would not be easy, it is quite a difficult problem on it’s own.

**2.5.2 Proposed extensions**

**Surrogate mines.** In some scenarios it might be convenient if one type of mines could be used as a surrogate for another type. Newer mines are most often prefered to older ones, so the user might want to assign the best type of mines to all minings in a scenario and then let the planner decide where the best mines and where the surrogate mines should be used.

**Other ways of using nodes.** In some scenarios the trucks could be used more efficiently if they where allowed to transport the mines in shift. It is easy to find examples where trucks could cooperate, picking up mines and then carrying them somewhere where they are temporarily stored, to achieve swifter transportation.

**Doing more than one job at a time.** The ships only do one job at a time. This effectively prevents the planner from being able to fully use the capacity of the ships. It is easy to figure out that the ships could load mines for a lower prioritized mining whilst waiting for mines for a mining of higher priority to arrive.

**Interaction with the user.** It is hard for the planner to know if it has chosen the best set of minings. To circumvent this drawback, it could generate the combinations of minings...
possible to mine, and then asks the user which combination he wants planned. This would not only affect the planner but also the client (and the server which has to support the communication).

2.5.3 Performance

We have no way of checking how close generated plans are to the optimal solution, but intuitively the plans seem to be good. We have not found any previous work that precisely corresponds to our problem.

2.5.4 Other algorithms

The algorithms used are by no means the best ones there are. They are only sufficiently good. The problem with optimal planners is that they tend to be inefficient. In section 2.3 a short motivation for the usage of a two-step algorithm was given. A one-step algorithm on the other hand, could produce better plans. Being able to route the mines from stores all the way to the mining, the planner could greatly increase the usage of the resources. The resources could be used for many different jobs at the same time and the planner could even try to minimize the use of them.
Chapter 3

Process and database management


3.1 Introduction

One of our goals was to produce a distributed multi-user system. In this chapter we will describe how we achieved this goal. We will also discuss the consequences of the design-decision to implement the planner in PROLOG and the rest of the system in ERLANG.

To fulfill this goal we needed process management (Section 3.2), database management (Section 3.3), and integration of the system (Section 3.4). These components are collectively referred to as the server. The main responsibility of the process management is to make sure that all processes in the system stays alive. Database management is responsible of assuring the integrity of all persistent data. System integration provides communication between the processes as needed.

In this chapter a user refers not only to the person using the system, but also to the information about this user stored in the user database. All users have a unique user name.

An administrator is a special user who is allowed to add new users to the system, move processes, and so forth.

A client is a program with which a user can connect to the system. There are three types of clients in the system: the drawing tool (Section 4.3), the analysis tool (Section 4.4) and the system manager (Section 3.2.3).

In addition to the supervision of the system, the server is made up of: The administrative server (AS), the administrative database (ADB) the scenario database (SDB), and the name server. The relationships between these components are described in Figure 3.1.
Figure 3.1: Server structure, and the relationships between all clients: the system manager (SM), the drawing tool, and the analysing tool (both named Client in the figure). There can be several instances of the components marked with a star (*), on the same or on different machines. All components in the same square must run on the same machine; components in different squares can, but do not have to, run on different machines.
3.2 Process management

3.2.1 Requirements

The main issues in process management are to make the system robust and efficient. The system can be distributed on several hosts which provides means to achieve these goals.

In order to make the system robust there must be a way to supervise the processes in the system. A process that exits abnormally should be restarted at a suitable node. If a machine crashes, then all processes on that node should be restarted on another node. For this to work, processes must be able to log themselves, so that they can be restarted with their last logged state. When a process is restarted, this should not affect other processes in the system.

To make the system efficient, we need a good strategy for selecting nodes to create new processes on. This strategy should take into account that some processes may not be able to run at all available machines. A graphical overview of the system that allows explicit planning of the distribution is also needed (this could also simplify debugging of the system).

In addition the system should provide some user management services and a way to shut down the system in a controlled manner. To simplify the treatment of processes we would like to make the fact that the system is distributed transparent for the rest of the system.

3.2.2 Design

There are essentially four processes concerning process management:

- The master process supervises the other system processes.
- The supervisor is a small process whose only job is to supervise the master.
- The process database contains information about all supervised processes in the system, described on page 26.
- The system manager provides several services concerning user management and explicit distribution planning, described on page 26.

System organization

The system is organized as a pool of ERLANG nodes which consists of a master node and a set of slave nodes. The slave node’s I/O is done via the master node. If the master node terminates, all the slave nodes also terminate. Therefore, the system manager is not allowed to take down the master node. The names of the hosts on which ERLANG nodes are started initially are specified in a resource file.
A statistics collector is running on each node, which measures the length of the CPU run queue (that is, the number of processes that are ready to execute). This gives a good measure of the future load of the nodes. Reports are regularly sent from the slave nodes to the master node about their current load. A process on the master node, the *pool-master*, uses these reports to maintain a list of nodes, sorted on a least-load-first basis. Queries may be made to the pool-master about which node that has the least load.

The strategy to select nodes for new processes is quite simple. If possible the master and the supervisor should be on different nodes. For all other processes the node which has the least load is selected among the available nodes for that process. Such a node is later referred to as the best available node. If a process is unavailable to a process for some reason, this can be specified in a resource file, the *module restriction file*. This file is consulted every time a process is created to find the available nodes for that process.

**Supervised processes**

All main processes in the system are supervised. These processes will always be restarted if they exit abnormally. Information about all supervised processes are stored in the process database. This information is used at process restart. Each process that is supervised is named with an unique abstract process identifier (pid) which it keeps even if it is restarted. Since the abstract pid of a process never changes there is no need to inform other processes if a supervised process is restarted.

When we use abstract pids, we want to hide the fact that the system is distributed. To achieve this we have used *global registration*. This facility implements *location transparency* for the supervised processes. That is, when sending a message to a process, the sender does not need to know at which node the receiver is located. Thus, we have the same functionality as we have with true pids in *Erlang*.

If a process goes down for some reason, there is a small chance that another process tries to send a message to it before it has been restarted. To guarantee that no messages are lost, we use a special send command when sending messages to supervised processes. If the receiver is a supervised process that is not running at the moment, the sender will have to wait until the receiver has been restarted.

To be able to treat all supervised processes in a uniform way, some demands are set on processes that should be supervised:

- At least two functions must be provided:
  - A function that is used to start the process initially.
  - A function that the master can use to restart the process.
- Processes must be able to receive at least two messages:
- A move message, which is sent from the master process when the receiver is about to be moved.
- A close message, which is sent when the system is about to be taken down. The receiver should exit, but it will not be restarted.

### 3.2.3 Implementation

**Global registration**

The global registration facility is implemented by having a relay process running at each node in the system. Each relay process maintains a dictionary where abstract pids are associated with their true pids. A message to an abstract pid is always sent via the local relay process. If the abstract pid is unknown, the local relay process queries all other relay processes in the system for the true pid. If such a pid is received, the message is sent to it. If not an error message is printed.

An abstract pid must be unique. To generate the abstract pids we use a unique name server, described in Section 3.3.3.

**The master module**

The master module starts up the system initially. When the system is running the master provides services to the system manager and restarts processes that for some reason go down. The master also provides a command which shuts down the system in a controlled manner.

After the system has been initiated, the master’s main concerns are:

*Process restart.* At process restart, the master uses information from the process database. If such information exists, the master restarts the process at the best available node. Otherwise, the master does nothing. The supervisor and the process database are treated specially, since they are the only processes which does not take their state from the process database.

*Process exit.* If a process exits in a controlled manner, its entry in the process database is removed. In case of an abnormal exit, the process is restarted.

*Node failure.* If a node goes down, all processes on that node is located via the process database. These are then restarted one by one.

*Process move.* The master can receive a request to move a process from the system manager. If the target node is among the available nodes for the process, the master first shuts down the process and then restarts it at the target node. If the target node is not an available node, an error message is displayed. The master is not allowed to be moved.
Stop node. If a node stop is requested from the system manager, the master first check that the node is not the master node. If it is the master node an error message is displayed. Then the abstract pids on all processes that reside on that node are retrieved from the process database. If there exists a process on the node for which no other node is available, the master sends a warning to the system manager. Before stopping the node, all processes are moved to other nodes in the system.

Closing the system. When the master is requested to close down the system, all system processes must be closed first. A list of all system processes is retrieved from the process database. Some processes have reason to close other processes themselves; these are placed first in the list. After closing a process the master waits for the corresponding exit message and removes the entry in the process database when it arrives. Before closing the next process in the list, the master flushes its message box for exit messages. This should be done because the closed process might have closed some other processes itself.

The process database

The process database is used to store live information about processes in the server in a secure way.

Requirements. The database must be able to store persistent information. Read operations are very usual and must be done in a fast way. There should be fast read and write operations with either the true pid or the abstract pid as the key.

It is very important that the database can be restarted from its previous state if it has died of some reason. A field in the database should contain at least the following: the true pid, the abstract pid, module name, logged state information, and the node the process is running on.

The database must also provide functionality for a process to update its logged state.

Design and implementation. The database is based on two hash tables, one with the true pid as the key and one with the abstract pid as the key. The second table has the function of an index and refers to the first table. This allows both read and write operations to be done with either the abstract pid or the true pid as the key. There is also a copy of the database on disk that is updated after each change of the database.

The system manager

The purpose of the system manager is to provide a user-friendly and intuitive environment for the system administrator when handling the system and its users.
Requirements.

- The system manager should only be accessible by the system administrator.
- There should be a view of current system messages.
- There should be a way of removing a node from the system. This should be done in a robust way so that any process running on the node is moved to some other node in the system. There should also be some way of adding a new machine with a running node to the system.
- Dynamic process migration should be supported with a ‘drag-and-drop’ interface. The purpose of this facility is to provide for the system administrator to supervise and distribute the main processes over the machines in the system (to move processes away from machines with heavy load).
- The node handling (see below) should also be done in a easy way and it should be clear which processes are running on which machine.
- Load statistics should be provided for all machines in the system.
- Editors for the resource files should be provided.
- Functionality for adding and removing users and changing passwords should be provided.

Design and implementation. To meet these requirements we have used a graphical environment described below.

Base window. To get a good overview of the systems dynamic state we have a base window containing a system message subwindow and a list of all running nodes.

A node is represented as a table of the processes running on it. Above each node table is a load window with a graphical representation of the load statistics for that machine. This gives a good overview of the systems dynamic state. The load statistics is the load given by the Unix command `w -u`. See Figure 3.2 for an example of the base window.

The table representation of the machines is also used for the process migration. This provides an easy way to move a process from one machine to another via a ‘drag-and-drop’ interface.

Node handling window. Adding and removing nodes dynamically at runtime are done in a separate window where two lists are displayed, one list of machines with nodes already running, and one list of available machines. See Figure 3.3 for an example.

User management window. Adding and removing users, and changing passwords are done in a separate window. This window contain among other things a list of all users in the system. Users that are logged on may not be deleted from the system. User passwords
Figure 3.2: The base window of the system manager with four different machines running. The process distribution follows the load — no processes at all are running on mamba.csd.uu.se since it has a high load. The tables for the machines have entries for all main processes running on them.

Figure 3.3: The node management window for adding and removing nodes dynamically at runtime.
may be changed by the system administrator without knowing the users old password. The system administrator’s own password may also be changed but the old password must be known.

Resource files. There are four different configuration files: two host files, the name server file and the module restriction file. Each of these can be edited with a resource editor. In figure 3.5 is a example of the resource editor for the host files.

- There are two host files:
  1. A file with the names of the machines the system will start running on.
  2. A file with names of machines on which new nodes can be started. When using the dynamic node handling tool for starting new nodes, the available machines is taken from this file.

When the resource editor is used to add machines to the second host file it will examine the local net to see if it is a existing machine and make sure that the name is complete.

- The name server file contains machine names where a name server may be started and a UNIX port number for each machine. The file must contain at least one machine on which the system will run. The file may contain any number of machine names, but only one name server will be started (see discussion of the nameserver on page 39).

- The module restriction file contains information about where hardware-dependent modules in system can run. For example, the planner module which is written for a PROLOG system that is compiled to run only on HP-UX.
3.3 Database management

We have chosen to divide the users work into logical components. A project is the topmost grouping consisting of a scenario, a map, and possibly some solutions. A solution is a scenario and a list of plans. This scenario is a locked version of the project’s scenario, it cannot be changed.

To make the creation of a project easier we have made it possible to create projects from templates. A template is a map and a scenario. A map is a bitmap and information about the size of the map in the real world. This means that a user can save the scenario and the map of a project in a template, and then create new projects from this template. For example, when the user has placed all stores, quays, and roads on the map, he can save it as a template and then experiment with different placements of minings, trucks, ships, and mines.

A project can be active or passive. It is active if there is at least one user working on the scenario of the project, otherwise it is passive. Several users can work on the same scenario at the same time. When one user modifies the scenario of a project, all clients working on the same project will be updated.

The database system consists of three components:

- The scenario database (SDB)
- The administrative database (ADB)
- The administrative Server (AS)

There is one SDB for each active project; it handles all request concerning the project’s scenario. The ADB consists of several databases with information about projects, users, templates, etc. There is one AS in the system, it coordinates all databases and clients.
### 3.3.1 Requirements

The scenario database

The SDB should be distributed, allow concurrent editing, and assure that the data is consistent and persistent. Speed is also essential, since this database will be used for graphical editing.

The functionality of the SDB should make it possible to store and edit information about the scenario.

The administrative database

The ADB should make data persistent. It could also provide some level of authentication, but we have not emphasized this aspect.

The ADB should supply functionality to save and edit projects, users, templates, maps and types. A type is a description of components, such as mines, or resources, such as trucks.

The administrative server

The AS should make sure that all user requests are handled in the right order and keep track of users and projects.

### 3.3.2 Design

The scenario database

This database is stored at a scenario server. Every client connected to a project has a local copy of the database for faster read access.

If a client wants to edit the database, this will be done at the original copy in the scenario server. To guarantee consistency between clients we have used a multicast protocol; all modifications in the database are sent to all clients by a dedicated multicast process. This process must guarantee that all clients get these messages.

We do not need any transaction handling, since there are no relative updates. That is, we never need to increase or decrease any attributes. If two or more clients tries to change the same attribute of the same node at the same time, then the value given by the last client will be stored. (There will always be a last client, since all updates must be sent to the scenario server, where an ordering will be imposed on them.) If a client tries to modify a node (or edge) that some other client is deleting, then the modify request will be ignored by the server.
The administrative database

The ADB is divided into a set of databases: a project database, a user database, a template database, a map database, a type database, and a database of connections.

The project database is designed as a hierarchical database, divided into three levels of granularity: project, solution, plan.

To make a question to a lower level than project, for example to find a solution to a specific project, the key for the project and the key for the solution are needed.

All the other databases have a flat structure.

Each question to any of the databases is a RPC to the generic database engine (Section 3.3.3), which also guarantees persistence of the database. Whenever a new record is created in a database, a unique name is created.

The Administrative Server

The AS supplies the following functionality:

- Allow the drawing tool client (Section 4.3) and the analysing tool client (Section 4.4) to connect and log in through channels (Section 3.4).
- Handle requests from these clients.
- Handle requests from the system manager (Section 3.2.3) through RPCs.
- Start and stop the scenario database.
- Handle requests from the scenario database through RPCs.

3.3.3 Implementation

We have based our databases upon the ERLANG term storage (ETS), which is supplied with the ERLANG run-time system. ETS provides the ability to store (extremely) large quantities of data in an ERLANG system, with constant access time. In ETS data is organized as a set of dynamic tables (hash tables).

The scenario database

The scenario database is implemented with five parts. A central database, several local databases, a database server, a database log and a multicast module.

The database server

The database server is connected to several local databases and guarantees that they are
kept consistent. When the database server receives an edit request from a local database, this request is first executed in the central database. If the request is granted by the central database the request is multicasted to the local databases. All granted requests are recorded on a database log. This log is used to restore the state of the database server after a crash.

The central database
To minimize the traffic on the channels between the central database and the local databases, we have decided to make the databases representation-dependent. The representation makes it possible to send single update requests that otherwise would have required several different requests. For example, if we remove a node all connected edges should also be removed. The update mechanism in the central database and the local database must therefore be similar.

The local database
When the client requests information of any kind the local database provides this information without contacting the central database. When the client wants to edit the database the request is forwarded to the database server via the channel. When the local database gets a granted request from the database server it executes the request and check the subscribe table. Subscribe is a special feature in the local database that allows the client to associate a function with a type of update. These associations are stored in the subscribe database. Some types of updates then results in a function call.

The multicast module
Multicast in Erlang is implemented using a process group that distributes a message to all processes in the group. In Erlang all messages sent from one process to another are guaranteed to be received in the order sent, and therefore consistency is gained by letting all requests sent from the database server pass through the multicast on the way to the local databases.

A multicast of this type must guarantee atomicity if it should be able to guarantee consistency in the local databases. To solve the atomicity problem that arises when a multicast crashes, we let the multicsats terminate the contact with the local databases. When the connections with the local databases are terminated they will request a new session and the current state of the central database is sent to them. In this way all local databases are kept consistent even after a crash.

The administrative database
The ADB provides two modules that are used by a project database. In addition, the ADB contains five databases holding maps, templates, types, users, and channels.

The generic database engine (GDBE)
This is a package to provide robust databases in an efficient way. Each database that uses the GDBE has two files attached, a data file and a log file. The log file consist of a trace of
events in the database; after a fixed number of transactions the log is emptied to avoid long restart times. When the database is started, the log file is read into the database. When the database is stopped in a controlled manner, the log file is emptied. The actual database is implemented using ETS. There are five types of transactions possible in GDBE: insert, delete, lookup, search and debug. The transactions are performed by message passing from the calling process.

The unique name server
This is a database that uses the GDBE to provide a unique-name service. A module that needs the service sends a unique key (for example the name of the module) and then gets a number in return. All numbers can be reused if needed. Each record in the database consist of a key, the next number and a list of numbers to be reused.

The project database
The project database is implemented using three GDBEs, one for the project top level data, one for the solution data and one for the plans attached to the solution. At the project level is information stored about:

- identification key of the entry
- user defined name
- process id
- address of the SDB (if active)
- working directory
- attached users
- scenario
- a flag that marks if the project is active or not
- a key to the map connected to the project

Stored at the solution level is information about:

- identification key
- user defined name
- scenario
- working directory

At the plan level the following data is stored:
• identification key
• user defined name
• comment
• filename of the plan

The actual plan is stored on file, since it can get quite big.

*The map database*
This database stores the filenames of the maps and information about the size of the area the map covers. Each map is made up of two bitmaps, one for the full-size image and one for the overview image.

*The templates database*
This database stores named scenarios and references to maps in the map database.

*The types database*
This database stores information about each resource and component type. For resources it stores the loading capacity and speed. For components it stores their sizes.

*The user database*
This database keeps track of the information attached to each user, password, real life name, attached projects, active projects (if any) and which channels the user is logged in to (if any).

*The channel database*
This database keeps track of the connection between login channels and users.

**The Administrative Server**

The AS starts all the administrative databases and a socket server (Section 3.4.2). The address and the socket number are sent to the name server in order to enable clients to connect. When a client connects to the AS, the channel it connected from will be stored in the channel database. After a client has connected, it has to log in with a username and a password in order to get requests granted. The username will be stored together with the channel in the channel database, if the password matches the password in the user database.

When a client is connected and a user is logged in, the AS will grant requests concerning the project database. If a user wants to open a project that is not active, that project will be activated and a SDB will be started. The address of that SDB will then be returned to the client so that it connect directly to the SDB. If, on the other hand, the requested project is active, the address of the project will be looked up in the project database and returned to the client. In either case the name of the project and the channel of the client will be added to the user in the user database, and the user name and the channel to the
project database. When a user closes a project this information will be removed. If this is the only user connected to the project, the SDB will be terminated.

A user can create new projects and templates, add other users to a project, get solutions and plans to analyze, and edit the type database.

The SDB can get or save its scenario, create solutions and save plans. It can also request information from the type database.

The administrator can connect to the AS via the system manager. When he is connected, he can get a list of all users in the system, add or remove users, and change their passwords. After a request to get a user list, all subsequent user logins or logouts are sent to the system manager.

### 3.4 System integration

#### 3.4.1 Requirements

**client-server integration**

The requirements for how the clients should be integrated with the server are:

- The client should be provided with an easy to use, high-level interface to interact with the server.

- Migration of the server between hosts should be invisible to the client. The client should not notice anything except for a possible delay in service.

- The communication should have sufficiently short response times to allow for distributed editing of scenarios.

- Communication primitives should easily integrate with ERLANG’s message passing paradigm.

- The design and implementation should allow for clients to be written in languages other than ERLANG.

**planner-server integration**

The planner-server interface should meet the following requirements concerning start of the planner and exchange of data.

- The server should be able to start a planner. If so needed it should be possible to start it on a remote host.
• The server should monitor the planner and restart it if necessary.

• When a server is started, a scenario should be sent to the planner, and a contiguous flow of plans should then be received.

3.4.2 Design

The possibility of clients written in languages other than ERLANG pervades the entire design of the client-server interface. In practice this implies the use of the IP protocol. Since the communication is full duplex and clients are connected for a long time, the reliable services of TCP are preferred over UDP.

Messages sent between the client and the server can be classified into two types: administrative messages, such as logging in and list available works etc., and editing messages concerning updates of scenarios. Since these two message types concern different parts of the server, and since we have (relatively) hard response time requirements for the latter, we use an approach where the client has two channels to the server. Editing messages can then be sent directly to the process handling the distributed editing of that specific scenario, thus removing the need for the message to be routed through several processes in the server. More detail on the distributed editing is given in section 3.3.2.

Protocol

Since communication has to be independent of language, a protocol for transmitting data is needed. This protocol describes how messages are converted to and from binary format. Apart from having a set of fixed messages the protocol also allows for transmitting general messages. These general messages are tagged (possibly recursive) data types. This makes it possible to encode arbitrary data structures.

The ability to send arbitrary messages is extensively used, since it makes for a flexible system development.

Channels

A channel module insulates us from the tedious work of socket programming. It exports primitives for opening and closing channels as well as sending and receiving messages.

A process that opens a channel waits for the other party to connect. The process that is accepting connections can either specify a port for the connection or let the operating system choose a port to use. The connecting process then have to specify the IP address as well as the port to connect to.

When a channel is established between two processes, messages can be sent in either direction. The messages are encoded according to the protocol described above.
**Name Server**

The name server has one task, knowing the address of the administrative server. This is necessary since the server is not located on a fixed address. A client first has to establish a channel to the name server and request the address of the administrative server before connecting and logging.

The name server is located on one of a set of addresses specified in a resource file. When the name server starts it tries to accept connections on each of the addresses specified in the resource file until it succeeds. This scheme allows for a specified host to be unavailable. When a client connects to a name server it tries the addresses in the resource file in the same order as the name server until a connection is made.

**Client Interface**

Each service the client wants from the server corresponds to a message. Each of these messages is wrapped into a function that handles the communication with the server, relieving the client from these chores.

Examples of functions offered to the client are: login, list works, list solutions and open work.

**Planner Interface**

The decision to implement the planner in another language than the server, Prolog, requires an interface that can handle the communication with a process external to Erlang.

All communication between the server and the planner is made through the same interface, and functionality for sending and receiving messages in both directions is implemented. Messages are encoded using the protocol described above.

### 3.4.3 Implementation

**Protocol**

The protocol is implemented as collection of mutually recursive functions that parses objects and returns a list of bytes (integers in the range 0–255).

**Channels**

Channels are implemented using the Erlang socket library. Each channel consists of two Erlang processes (one at each end, called the channel processes) connected with a socket. The channel processes encode outgoing and decode incoming messages. Incoming
messages are sent to the owner (the process that created the channel) as an ERLANG message. Outgoing messages are sent to the encoding channel process, which sends it over the socket.

**Client Interface**

An ERLANG process in the client, the server-interface process, handles the connection and communication with the server. A client-interface function sends a message containing the function name and arguments to the server-interface process. This message is then sent over the channel to the server where it is handled in the administrative server. The response from the server is then returned as the result from the function.

The server-interface process is also responsible for starting the local database interface when a new work is opened. See section 3.3.3 for more details.

**Name Server**

Each row in the name server resource file specifies a host and a port. The name server tries to start a process on one of these nodes. When it succeeds that process will attempt to accept connections on the port that is associated with the node it runs on.

When the server is moved it sends a message notifying the name server of its new address (host and port). When a client connects to the name server it gets the current address of the server.

**Planner Interface**

Each time a scenario database receives a planning request a planner interface is created on the node the planner shall run on. The planner is started with the ERLANG primitive open_port. The scenario is then sent from the server to the planner through the planner interface.

The planner interface receives a contiguous flow of plans from the planner and relays them to the scenario database. The interface is also responsible for monitoring that the planner is alive, notifying the server otherwise.

**3.5 Discussion**

The use of ERLANG made it quite easy to construct a distributed system. However it is still very hard to construct a system that guarantees the integrity of data in all possible situations. We believe that we have the basis for such a system, and that we know how to make it; and with a little more time we could have made our system even better than
we did. We now know the importance of a well-planned, deadlock free, hand-shaking procedure for both startup and shutdown. As it is now, we know of some, very unlikely, situations where some data could be lost. For example, if the file system is full or if some key processes should crash at exactly the wrong time. Still, we can say, with confidence, that this is a really stable and robust system.

3.5.1 Optimizations

We have considered (but not implemented) two optimizations, concerning the reduction of network traffic: a complete local database for single users, and coded commands on channels.

If there is only one user connected to a project there, really is no need to send the update requests all the way to the scenario database server, since all requests will be granted. Therefore it would be possible to make a local database capable of both reading and writing and supply a method to automatically make use of this database when there is only one user connected.

The requests and answers that now are sent over the communications channels as strings of bytes, could be coded using only one or two bytes, reducing network traffic drastically.

3.5.2 Extensions and Omissions

There are a lot of possible extensions, we have only implemented those features that are most important. The administrative database could be greatly enhanced, for example with more information about users and projects. Some features, such as the users real name, are not fully implemented.

The security of the system is very low, the passwords are not even encrypted, but we felt that this was not an important aspect, since the system is supposed to be used on computers and networks that already are secure.

The most annoying restriction of the current system is the need to run the clients on computers that have the same file system as the server mounted. This is due to the need for the clients to read map bitmaps from the file system. It would be possible to send these bitmaps over the channels, instead of their filenames as we do now. If this improvement would be implemented, then it would be possible to use clients running on machines on completely different file systems; it would for example be possible to run a client on a PC with Linux over a telephone line.
Chapter 4

User interface

By Ola Strandberg, Robert Sundström, Fredrik Larsson, Johan Tjäder.

4.1 Introduction

The user interface should provide facilities for the design of a scenario, an interface to the planning and facilities for the analysis of the resulting plans.

The scenario is the set of components, resources, stores, destinations and their attributes. In our case, components are mines, resources are trucks and ships, stores are stores in which mines are kept and the destinations are minings. Mines have a given assembly time, trucks and ships have speed and load capacity, etc. For a fuller discussion, see section 2.1.

The design of a scenario is done by placing stores, quays and minings on a map, connecting them by roads and finally, by placing the resources and defining the need for components in the destinations.

The planning stage results in a solution that contains the scenario and possibly several incrementally improved plans. A plan is a series of event traces for each resource. The analysis of a plan consists of deducing when the laying of a specific mining was completed, how many mines were left in a specific store at any given time, when the laying of mines was complete, etc.

The tool that was constructed consists of several parts: a drawing tool for designing the scenario, an analysis tool for analyzing the plans, and an interface to administrative functions of the system.

In the following discussion, preplanning activities are carried out when designing the scenario and postplanning activities are carried out during the analysis. For the preplanning activities, the drawing tool is used; for the postplanning activities, the analysis tool is used.
4.2 Requirements

The requirements of the user interface are in many aspects those of the whole system, although much of their functionality is implemented in other parts of the system. Examples include:

- The system should offer persistent initial states, i.e. changes to the scenario of the system should not be lost in the event of a communication failure or other system error. This must be combined with explicit naming of initial states.

- The system should offer cooperative editing of a scenario by several users. The system can thus be used in teacher/student situations.

- The system should not allow unauthorized access.

The first of these requirements has no impact on the design of the actual user interface, except for the naming of initial states. The second requirement has a larger impact. The drawing tool must be prepared not only for local modifications, but also for nonlocal updates of object attributes. Apart from this, most of the details are implemented in the server. A login dialog suffices to satisfy the third of these requirements.

Functional requirements of the user interface are:

- The possibility to use a map in the form of a background image in the drawing tool.

- Zooming and panning of the map should be possible.

- ‘Real world’ measurements should be used in all communications with the user.

- The possibility to save initial states as templates.

- The possibility to save a project with several cooperating users as a new personal initial state for continued editing alone.

- The possibility to lock other users from a personal project.

- The system should conform to sample data that was given in a scenario prepared by Björn O Bergström, Telhub [2].

- Cooperating users should be able to communicate to exchange ideas, etc.

- For cooperative editing, the emphasis is on cooperative. Lost updates are ignored; the last person to ‘commit’ an update ‘wins’.

Nonfunctional requirements:
The user interface should provide an intuitive interface to drawing functions, i.e. similar to 'standard' drawing tools.

- The drawing tool and analysis tool should have consistent interfaces.
- The system should use Swedish in all communication with the user.
- The system can assume color monitors but must work on greyscale monitors with at least 16 levels of grey.

4.3 Preplanning activities

The preplanning activities consist of opening or creating a project and defining the scenario.

4.3.1 Human-computer interface design

The drawing tool consists of a main window that contains a menu bar, canvas, tool palette, properties window and an overview window. The client area is everything in the main window below the menu bar.

The canvas is the drawing area, the tool palette contains shortcut buttons with indicators for the different drawing tools, the properties window displays messages and information about selected components, and the overview window contains a scaled down copy of the canvas used for panning and zooming of the canvas contents. The client area is laid out with the canvas on the left side with vertical extent all the way to the bottom of the window. The tool palette is placed in the upper right corner of the client area. The properties window is placed in the mid right part and the overview window in the bottom right part.

The objects that are to be drawn are stores, quays, minings, roads, waterways and road crossings. Tools for drawing these objects are provided and commands for selecting these tools are duplicated in the tool palette, the menu and by clicking the right mouse button, as will be discussed later. Stores, quays, minings and road crossings are all nodes in the sense that they are connected by edges (roads and waterways) and they can hold resources and components. These objects are further described in section 2.2.

The tool palette

The tool palette has the following tools:

- Select tool
- Store tool
Figure 4.1: The drawing tool with a scenario. Stores are indicated by filled circles, quays by filled squares, minings by opaque filled rectangles, and crossings by opaque circles.
• Quay tool
• Mining tool
• Edge tool
• Crossing tool

The select tool is used to select and move objects. The other tools are used to draw their corresponding objects. The buttons in the tool palette have indicators to show which tool is selected at all times. The buttons have standard symbols taken from [2].

Use of the mouse

Mouse behavior depends on what tool is selected. When any of the node tools (store tool, quay tool, etc.) is selected, a click of the left mouse button places an object on the canvas. When a store, quay or mining is drawn, a dialog pops up with default values for name, id, etc. allowing the user to enter the number of resources and components. These values can also be changed by double-clicking on the node. If the mouse is clicked inside another node, the action is suppressed rather than placing another node on top of the one that was clicked.

The other tools (edge tool and select tool) have more complex behavior:

Edge tool An edge can only be drawn between two nodes. An edge is drawn by pressing the left mouse button inside a node, dragging the mouse with the button pressed and releasing the button inside the destination node. If the edge tool is selected, mouse clicks outside nodes have no effect. If the mouse button is released outside a node when drawing of an edge is in progress, the edge is erased. An edge can be of two types: road and waterway. A waterway can only be drawn between a quay and mining, between mining and mining, or between a crossing and a mining. Other edges are roads. Edges are displayed in grey during drawing; once the type of the edge has been determined, the color is changed to blue to reflect a waterway or black to reflect a road. The type of the edge can be changed by the user.

Select tool The select tool is used to move and select objects. An object is selected by clicking the left mouse button inside the object. Several objects can be selected by clicking inside objects with the middle mouse button. Selected objects are deselected if the left mouse button is clicked outside any object. A selected object is indicated by a red outline. When an object is selected, the properties window is updated to show information about the selected object. If several objects are selected, the properties window shows ‘multiple selection’. An object or selection is moved by clicking the left mouse button inside the object or one of the selected objects and dragging the mouse with the button pressed. When a selection is moved, all selected objects are
The movement is stopped when the mouse button is released. Edges cannot, once drawn, be moved to other nodes. When a node is moved, all connected edges are ‘rubberbanded’ and follow the node.

The right mouse button

The right mouse button is used to provide context sensitive commands, that is, commands that depend on the context in which they are applied. When the right mouse button is clicked outside an object and there are no selected objects, a menu with a textual representation of the tool palette is displayed. If an object is selected, a menu with the single item ‘Delete’ is displayed. If no object is selected and the right mouse button is clicked inside a node, a menu with the items ‘Edit’ and ‘Delete’ is displayed. Selecting ‘Edit’ brings up the same dialog that was displayed when the object was drawn for further modification. If the right mouse button is clicked on an edge, the menu that is displayed allows changing of the edge type and deletion of the edge.

The properties window

The properties window displays information about selected objects. When a new object is created it is also updated to display information about this object.

The overview window

The overview window is used to zoom and pan in the canvas. Typically a scaled down version of the background map is displayed. The currently shown portion of the whole map is indicated by a rectangle in the overview window. Panning is done by dragging the rectangle in the overview window or by clicking the left mouse button where the new center point should be. Above the overview window, two buttons for zooming are available. Zooming is done by pressing these buttons or by pressing ‘+’ or ‘-’ on the keyboard.

4.3.2 Subsystem design

An object oriented approach was taken from the start. Rumbaugh’s OMT model [?] was used to prepare a class diagram of the application (in this discussion, the drawing tool). The application object maintains a list of documents and dispatches commands to other objects in the program. The unit of data that the user works with is a document (a project). The document maintains, loads and stores its data. The user interacts with a document through a view (the canvas) of the document. A view displays its document’s data and takes mouse and keyboard input which it translates into selection and editing actions. Objects in the user interface, such as menus and buttons, send commands to the documents, views and other objects in the application. These objects carry out the
commands. The document/view architecture is especially appropriate in our case since there will be multiple users working on the same document.

When the user modifies data through the view, the view notifies the document. The document in turn tells all of its views to update their displays with the new information, and the views respond by redrawing the document.

Other objects in the application represent the objects of our application world: minings, stores, quays, road crossings and roads and their drawing tools. Mouse input is directed to these objects for local handling of events.

**The application object**

The application object is split between the server and the client. Management of documents is done at the server level and management of views is done at the client level.

The application object is responsible for creating the other user interface objects such as windows, menu bars, etc. It is also responsible for routing messages between the other components.

**The document object**

The actual implementation of the document object is done in the server, with just a wrapper for meta data retrieval in the client. To accommodate several users, the ‘subscription’ mechanism described in section 3.3.3 was used. When an object is created, a subscription to changes of its attributes is added. If another user updates the object, the local document will be notified. The subscription mechanism was also used to receive notifications about added objects, removed objects and finished plans.

**The view object**

The view handles user interaction. The view’s main task is receiving mouse input and redirecting this input to the correct object in the view. Other tasks are coordinating zooming and panning in the map and recreating a scenario from the database when loaded.

**The drawing tool objects**

The drawing tools were designed as objects. A tool is selected at all times and the view delegates mouse messages to the selected tool.
4.3.3 Implementation

Implementing objects in ERLANG is naturally done by treating a process as an object. (For discussion, see section 4.5.1) The application object is one process and the view one. The document object is merely an abstraction of the database that is implemented in the server and is not discussed here.

4.3.4 Human-computer interface implementation

The human-computer interface was implemented with the ERLANG graphics system, gs. Gs is implemented as a portable front end written in ERLANG, and the vendor plans to write several back ends for different operating systems and machine architectures. Gs is at the time of this writing released in an alpha version, built on a Tcl/Tk back end.

The functionality of gs is somewhat limited and there were a number limitations that needed to be overcome:

- displaying color bitmaps
- scaling images
- setting of certain drawing object attributes, such as hatched fill
- zooming and scrolling canvases
- setting viewport and origin of a canvas
- creating pop-up menus
- weak error handling

Since gs interfaces to Tcl/Tk it was very easy to customize the behavior by writing Tcl scripts to implement new functionality or by using a built-in facility for sending Tcl commands directly from ERLANG. Tcl scripts were written to handle all of the above but the last, and further functionality in the form of tables was built in by extending wish, Tcl/Tk’s shell, and writing an ERLANG interface for tables. For the last point, gs was modified to crash with an error message rather than go into an infinite loop as it was originally.

Furthermore, when the canvas was zoomed, hit testing of objects was difficult since gs could only handle pixel coordinates relative to the upper left corner of the displayed area of the canvas. Tcl/Tk has functionality built in to customize the behavior for all objects of a given type, so zooming, scrolling, hit testing and scaling of objects were implemented wholly in Tcl/Tk for all node objects. For nodes we wanted the symbols drawn to have the same size regardless of the zoom factor, which would have been difficult to implement in gs.
A generic module were written to handle user input and messages to users. This module implements the objects MessageBox, StringBox, and Browser. MessageBox shows a message to the user in a window, StringBox prompts the user to input a string of text and Browser shows a list of alternatives and prompts the user to select one. All of these accept a title, a message and a list of prompts for buttons to be created. Optional ‘hook’ functions can be supplied to customize the behavior of pressing buttons in the dialogs.

The application object

The application object was implemented as a process. The application object ‘owns’ the menu and the tool palette, and handles view creation and administrative commands. Tool palette messages are simply passed on to the view, as are drawing related menu commands. Administrative menu commands such as ‘create new project’, ‘open project’ and so on are handled directly by the application object.

The view object

The view object was also implemented as a process. The drawing tools share a common loop except for the select tool that has behavior complex enough for a loop of its own. The current tool identifier is passed around in the loop at all times and when a message is received, the apply mechanism described in section 4.5.1 is used to route the message to the correct object.

When a gs object is created it returns an identifier, by which it must be referred for all subsequent manipulation. Since we need to accomodate several users, the gs object identifiers cannot be stored in the database; they will be different on different systems. Also, the database used in the analysis is read-only, which is another reason that the gs identifiers cannot be used. Fortunately, gs offers a way to register objects to give them names. The object identifier from the document was used to create unique, system-independent names that could be used to register the gs objects.

4.4 Postplanning activities

The postplanning activities consist of analysing the plans that are generated by the planning phase.

4.4.1 Design

The design of the analysis tool interface was deliberately kept the same as the drawing tool.
Instead of the tool palette, a list box is displayed with all incrementally improved plans. Since the drawing tool and analysis tools are separate, messages about new plans are not sent to the analysis tool, but only to the drawing tool that initiated the planning phase. To accommodate new plans, clicking in the plan selection list updates the list and adds new plans if any.

The information wanted from the analysis of a plan is basically ‘At what time was mining X laid?’ To accommodate this question, a slider was placed at the bottom of the window. The slider’s range is set to the time of the last event in the plan’s event trace. By dragging the slider, the current time is shown and minings change color as they are filled. A textual representation was also made available with tables listing all minings with their laying times and a table listing the stores and showing the number of mines of each type of mine in the store at the current time. An additional slider was placed in this window to allow update of the time directly.

From a plan, a schedule for each resource can be printed as well as the tables described above.

The application and view objects are basically the same on the design level albeit with less functionality in the view. The ‘real world objects’ (such as Store, Quay, etc.) have slightly different behavior since they cannot be edited during the analysis.

The document object also has the same functionality as the drawing tool’s document but keeps the current time internally and answers queries about mines by returning the value at the current time. The analysis tool’s document is not a project as in the drawing tool, but a plan, and hence also contains information about schedules for resources. When designing the scenario, resources are not seen as ‘individuals’ but just as numbers placed in nodes. When the initial state is preprocessed before the planning phase, resource objects are created that can later be analysed in the plan.

4.4.2 Implementation

The entire drawing tool was copied, stripped down and modified to accommodate the analysis of plans instead of creating scenarios. The main issue was rerouting the calls to the document object from the project object that is used during the creation of the scenario (that is a client/server activity) to the plan object that is to be analyzed (that is a local activity performed in the client). To avoid the overhead of checking at every call which database to call, the entire document object was also copied and modified to use the local plan database instead of the project database in the server. This had the implication that the integrated drawing/analysis tool that was planned was difficult to create. We had to accept two different tools.
Figure 4.2: The analysis tool, displaying the results of the laying of the minings. The results are represented by DD HH:MM:SS from start.
4.5 Discussion

The anticipated main issues of the implementation of the user interface were efficiency of hit testing and screen repaint on updates. As it turned out, Tcl/Tk is different from most graphics system implementations. Once a graphics object is created in a Tcl/Tk window, Tcl/Tk takes control of the object and redraws it when necessary. For instance, when moving an object, no erasing of the object at the previous position is necessary, since it is XOR’ed on the background. As for hit testing, a naive algorithm was used, simply listing all objects and applying a hit test algorithm on each in turn. Other approaches were considered, such as having a ‘network’ database sorted on coordinates, but were not tested.

Another efficiency consideration was the client/server structure. When a user was editing a scenario alone, as opposed to cooperating with other users, a ‘short circuit’ in the document object was considered to minimize lag times when manipulating the database. When for instance a node was moved on the screen, its new coordinates were not displayed in the properties window when the object was released since the updated coordinates were not written to the database quickly enough. This was instead implemented by a one-object cache in the document. Read operations always check the cache before a database read operation is done.

4.5.1 Object-orientation in Erlang

We implemented object-orientation in two ways. The first, natural approach is to treat processes as objects. This was done in the case of the application and the view. Message passing between these objects was implemented using the built-in ERLANG message passing mechanism.

For the implementation of the application world objects mining, store, etc., a different approach was used. First, the process approach was considered for these objects as well, but it was feared that that the number of processes that would be needed to hold all objects was too large. Instead, objects were implemented in separate modules with the message handlers as exported functions. The object identifiers in the database were constructed as tuples of `{datatype,unique-id}` and the datatype atom used to send messages to the correct object type via the ERLANG apply mechanism. The objects’ attributes are held by the database with the object identifier as the key. Inheritance was not simulated or attempted. Edge was considered a datatype of its own, while all node types, mining, store, road crossing and quay share all attributes.

An example may be appropriate. This is an excerpt of the hit testing algorithm:

```erlang
hit_test(X, Y) ->
    Objs = is:list(),  % Retrieve all objects in the database
    hit_test(X, Y, Objs).
```

52
hit_test(X, Y, []) ->
  {hit, no};  %% If we reach this, no object was hit
hit_test(X, Y, [H|T]) ->
  {ObjType, _} = H,  %% Get object type
  case apply(ObjType, hit_test, [X, Y, H]) of
    {hit, yes} ->  %% We hit this object
      {hit, H};  %% return it.
    {hit, no} ->  %% We didn’t hit it,
      hit_test(X, Y, T)  %% so recurse
  end.

Each datatype (module) has a function called hit_test that returns {hit,yes} or {hit,no} using an appropriate algorithm. The ‘objects’ read the data needed for local processing from the database.

This approach has the advantage of being efficient, but leads to code being duplicated in all modules having the same behavior. Superclass modules can be implemented and called from the subclasses where default behavior is wanted, but this was not used in the project at hand. Other OO-mappings to ERLANG, such as implementing a catch-all handler in subclasses that forwards unknown messages to the superclass, or even rewriting the ERLANG exception handler to forward unhandled messages, seem like trying to force ERLANG into being more object-oriented than necessary. We considered polymorphism and dynamic binding the most important features, not inheritance.
Chapter 5

Conclusions

Real-world planning problems are typically both complex and hard. They are complex in that they involve concepts that are difficult to model (e.g., production, consumption, and allocation of resources). They are hard in that the algorithmic complexity of the problems is typically high, since the solution spaces are very large.

The Plan95 system addresses these issues by employing modern software tools:

- The Erlang programming language facilitates modelling of complex concepts at a high level of abstraction.
- Constraint logic programming allows rapid development of programs that solve hard optimization problems reasonably well.

These tools greatly contributed to the success of the project.
Bibliography


