

MEDNET

Final Report for the Intelligent Systems Engineering Course 2025/2026

Pier Costante Babini

`piercostante.babini@studio.unibo.it`

Marco Morelli

`marco.morelli15@studio.unibo.it`

September 2026

MedNet is a two-tier multi-agent system that simulates a medical emergency network: a control centre, a fleet of ambulances and several hospitals that negotiate the admission of each patient among themselves and, internally, the assignment of a doctor. The system integrates BDI agents written in Jason/AgentSpeak, FIPA ACL communication, a generic Contract Net Protocol reused across two levels of negotiation, a tuProlog knowledge base shared between the Java side and the agent side, a STRIPS planner that derives care plans and role-based access control enforced on every environment action. The result is a system in which admissions are arbitrated by the control centre, a critical case can preempt a less urgent treatment and then be replanned while preserving the exams already performed, an ambulance can be rerouted during transport, and mutually exclusive equipment is coordinated by an arbiter with a priority queue and watchdogs.

Disclaimer. During the preparation of this work, the authors used GitHub Copilot to assist with code completion and support the English translation and the spelling review of the present report. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the final report/artifact.

Contents

1	Introduction	3
1.1	Goal and motivation of the project	3
1.2	Main contributions	3
2	Domain and core concepts	5
2.1	Ambulance dispatch and emergency department management	5
2.2	Why this is a good case for a MAS	5
2.3	Actors and roles	6
3	Requirements	8
3.1	Functional requirements	8
3.2	Non-functional requirements	9
3.3	Scope delimitation	10
3.4	Glossary	11
4	Course topics applied in MedNet	13
4.1	BDI agents and AgentSpeak(L)	13
4.2	FIPA ACL communication	14
4.3	Contract Net Protocol: negotiation and coordination	15
4.4	Knowledge representation and symbolic reasoning in Prolog	16
4.5	Classical planning with STRIPS	17
4.6	Security and access control: RBAC	18
5	Design	19
5.1	Architecture	19
5.1.1	Overall architecture	19
5.1.2	The percept/action cycle (Jason $\leftrightarrow$ Java)	20
5.1.3	The bridge to Prolog (Java $\rightarrow$ Prolog, Jason $\rightarrow$ Prolog)	21
5.1.4	Why clinical/organisational knowledge lives in Prolog	21
5.2	Infrastructure and world model	22
5.2.1	The discrete clock and listener ordering	22
5.2.2	The two spatial levels	23
5.2.3	Concurrency handling and lock acquisition order	23
5.2.4	Simulation termination	23
5.3	Modelling	24
5.3.1	Life of a patient (end-to-end path)	24
5.4	Interaction	26
5.4.1	The two-tier Contract Net Protocol	26
5.4.2	Bid formulas	28
5.4.3	Agent–environment interaction contract	31
5.5	Behaviour	35
5.5.1	Preemption	35

5.5.2	Renegotiation during transport	36
5.5.3	Mutual exclusion on shared equipment	37
5.6	Data and consistency	39
5.6.1	Planning: Prolog + STRIPS	39
5.7	Security	41
5.7.1	RBAC	41
6	Implementation and technologies	44
6.1	Technology stack and repository structure	44
6.2	Build, CI/CD, conventional commits	45
7	Testing and validation	46
7.1	Unit, protocol, and end-to-end levels	46
7.1.1	Unit level	46
7.1.2	Protocol level	46
7.1.3	End-to-end level	46
7.2	Practical constraints	47
7.3	What the tests do not cover, and why	47
8	Deployment and execution	48
8.1	Main scenarios	48
8.2	Build and run instructions	48
9	Self-evaluation	50
9.1	Pier Costante Babini	50
9.2	Marco Morelli	50
10	Conclusion and future work	51

1 Introduction

1.1 Goal and motivation of the project

MedNet is a multi-agent system that simulates the operation of a medical emergency network made of a control centre, a fleet of ambulances and several hospitals. The system models the whole journey of a patient: from the emergency call and the preliminary triage, through transport, hospital admission, diagnosis and discharge.

The main goal is to study how a set of autonomous agents can coordinate in order to allocate patients, staff and medical resources in a dynamic environment. The simulation involves several types of agent, each with specific responsibilities, interacting through percepts, actions and messages.

Two features of the domain drive the entire project: no agent knows the complete state of the network in advance, and that state changes while decisions are being taken. Both are discussed in Section 2.2, where they motivate the choice of a multi-agent system.

The simulation terminates when three distinct conditions are met; they are explained in Section 5.2.4.

1.2 Main contributions

The main contributions of the project are:

- a library implementing the *Contract Net Protocol*, reused both for the negotiation between the control centre and the hospitals and for the assignment of patients to doctors;
- an architecture based on BDI agents developed in Jason and AgentSpeak(L), integrated with a simulated environment written in Java;
- the use of tuProlog to represent clinical and organisational knowledge and to support the reasoning of the agents;
- a STRIPS planner to derive and dynamically update care plans;
- a cooperative *preemption* mechanism, which allows a less urgent treatment to be interrupted in order to handle a higher-priority emergency;
- a discrete tick-based simulation engine;
- a *replanning* mechanism that preserves the work already completed and recomputes only the stages of the care plan that are still needed;
- mutual exclusion over shared equipment through time-limited grants (*leases*), priority queues and *watchdogs*;
- the enforcement of role-based access control (*RBAC*) on every action agents perform in the environment;

- an automated test suite comprising unit tests, integration tests and end-to-end scenarios, run automatically by a CI pipeline on GitHub Actions.

2 Domain and core concepts

2.1 Ambulance dispatch and emergency department management

The simulated world is a territorial grid of 30×30 cells holding a number of fixed hospital sites and a fleet of ambulances. At unpredictable times and positions a patient becomes symptomatic and reports the emergency to a control centre, declaring the pathology. The centre assigns the nearest free ambulance to the call; the ambulance reaches the patient's cell, performs a preliminary triage on site, loads the patient and reports pathology and code back to the centre.

At that point the centre does not decide alone where to take the patient: it opens a negotiation with the hospitals of the network, each of which formulates a bid from the only knowledge it holds, namely its own position, its own free beds and its own specialisations. The winner reserves a bed and the ambulance receives its destination. On delivery the reservation becomes occupancy and the patient moves from the territorial level to the in-hospital one.

Inside the hospital the journey starts again on a smaller scale: a second assessment, a priority queue and a second negotiation, this time towards the doctors of the same hospital, at the end of which the winner does not run a hard-wired procedure but a care plan derived case by case. The complete path, from the two levels of negotiation through to discharge, is walked through in full in Section 5.3.1.

2.2 Why this is a good case for a MAS

The domain is a good case for a multi-agent system because none of the information a centralised decision would need is available in a single place, and because that information changes while the decision is being made.

First of all, the capacity of the network is not known in advance to anyone. The control centre perceives the position of the ambulances and the hospital sites, but receives *no* percept about the number of free beds of a hospital; each hospital knows only its own. The global state of the network is therefore not a variable to be read, but something that must be reconstructed through bids and refusals. This is exactly the situation the Contract Net Protocol is made to solve, and it is the reason why a centralised allocator would not merely have been a different choice, but one that is not feasible with the available percepts.

Furthermore, a *walk-in* event models an emergency external to the network that shows up at the door of a hospital: if it finds no free bed it takes away a reservation while the patient that bed was meant for may still be in the ambulance. A decision that is correct at tick t can therefore no longer be applicable at tick $t + 1$, and the system must renegotiate mid-simulation instead of failing.

Third, the entities of the domain are autonomous by nature and have distinct local goals: a hospital weighs its own load, a doctor their own working protocol, an ambulance its own mission. Modelling them as agents with their own beliefs, goals and plans, rather than as objects driven by a scheduler, makes the model faithful to the domain and turns coordination into an observable phenomenon instead of an implementation detail.

Finally, the domain naturally exhibits resource and priority conflicts: beds, equipment and doctors' time are scarce and contended, and a critical case must be able to overtake a less urgent one. Preemption, priority queues, mutual exclusion and replanning are therefore not features added as an exercise, but answers to constraints the domain imposes.

2.3 Actors and roles

The default configuration (`mednet_local.mas2j`) instantiates 24 agents over three hospitals. Table 1 lists the seven agent types, how many of each there are and their respective responsibilities.

Agent	No.	Responsibility
patient	5	Environmental stimulus: reports a single emergency, with a pathology.
control_center	1	Receives the calls, assigns the nearest free ambulance, runs the network CNP, handles reroutes and cancellations.
ambulance	3	Moves on the grid, performs the preliminary triage on site and transports patients.
hospital	3	Bids for admissions, reserves and releases beds, asks for a divert when it loses capacity.
triage_nurse	3	Secondary triage, priority queue, runs the in-hospital CNP, decides on preemption.
doctor	6	Bids on availability and specialisation, derives and executes the care plan, discharges.
equipment_manager	3	Arbitrates the priority queue over shared equipment and runs the watchdogs.

Table 1: Agent types, how many of each the default configuration holds, and their responsibilities.

The default hospital network, shared by every scenario, is reported in Table 2. The three equipment sets are identical (`ct_scanner`, `xray_room`, `ecg_station`, `lab`, `operating_room`); what differs is position, bed count and specialisations.

Hospital	Position	Beds	Specialisations
h1	(5, 5)	3	cardiology, general
h2	(24, 24)	2	neurology, general
h3	(5, 24)	2	trauma_surgery, general

Table 2: The default hospital network.

The naming convention for agents is not cosmetic: both the role, used by access control (Section 5.7.1), and the placement, used to compute the percepts (Section 5.4.3),

are derived from it. An agent named `doctor_h3_trauma_surgery` is a doctor, works in `h3` and is specialised in `trauma_surgery`, and these three facts are recovered from the name alone.

3 Requirements

3.1 Functional requirements

The functional requirements state what the system must do. Each one is numbered and comes with a verifiable acceptance criterion.

ID	Requirement	Acceptance criterion
FR1	A patient who becomes symptomatic reports the emergency to the control centre exactly once.	For every activated patient a single call is recorded, regardless of the order in which the activation percepts arrive.
FR2	The centre assigns a free ambulance to every call, choosing the nearest one, and never assigns the same ambulance to two calls.	In a run with several concurrent calls no ambulance is ever engaged on two missions.
FR3	The ambulance performs a preliminary triage on site and reports its outcome to the centre.	The triage datum is produced only after the assessment and is visible only to the ambulance that performed it.
FR4	The admission destination is decided by a negotiation among the hospitals, not by a centralised rule.	The winner is the hospital with the lowest bid; the losers receive an explicit refusal.
FR5	Every hospital formulates its own bid from the data it perceives locally, and from nothing else.	No capacity percept ever reaches the centre.
FR6	The bid accounts for the severity of the case: distance and clinical appropriateness weigh more as urgency grows.	Two patients with identical geometry and different codes are routed to different hospitals.
FR7	A hospital with no free beds does not take part in the award.	The saturated hospital answers with a refusal and cannot win the round.
FR8	If the assigned hospital loses the bed while the patient is still in transport, the negotiation is reopened excluding it and the ambulance is rerouted.	The rerouted patient is treated all the same and every reservation but the last one is cancelled.
FR9	Delivering the patient closes the network negotiation for good.	After delivery no round is awarded and no bed stays reserved.
FR10	A single real event reported by two sources must not multiply the negotiation rounds.	A double-trigger event opens exactly two rounds, no more.
FR11	On arrival at the hospital a secondary triage reveals the true code of the patient, who is then served in order of priority and, at equal priority, of arrival.	The queue model returns patients in (severity, arrival sequence) order and admits no duplicates.

ID	Requirement	Acceptance criterion
FR12	The doctor is assigned by a negotiation inside the hospital.	The available specialised doctor wins; every round assigns exactly one patient.
FR13	A red-code patient who finds no free doctor may preempt the least urgent treatment in progress.	The red case is taken over, the victim is handed back to the queue through <code>requeue_front</code> , which reinserts it at the head of its own priority class, and both are eventually discharged.
FR14	A preempted case resumes preserving the work already done.	The victim's second care plan is strictly shorter than the first.
FR15	The care plan is derived from domain knowledge, not hard-wired in the agent's code.	For every pathology a plan exists, and the expected plan is pinned by a test.
FR16	A piece of equipment is used by one doctor at a time, and access follows the priority of the case.	A single thread acquires the lock; an exam does not start without physical ownership of the equipment.
FR17	The system cannot deadlock over equipment.	No segment of a care plan ever holds two pieces of equipment, so no execution thread ever waits for one while holding another.
FR18	Every action on the environment is authorised on the basis of the agent's role, and whatever is not explicitly permitted is denied.	The full 7 roles $\times$ 17 actions matrix is checked exhaustively; unknown actions and agents are denied.
FR19	Every agent perceives only the portion of the world its role and its position allow it to see.	For every role both what it must see and what it must not see are checked.
FR20	The system recovers from resources held unduly (beds reserved and never claimed, locks held beyond expiry).	The recovery mechanisms work when exercised and never fire in a correct run.
FR21	A run recognises its own conclusion autonomously and freezes the final state.	The completion conditions mentioned above are all necessary for the simulation to complete.

Table 3: Functional requirements and their acceptance criteria.

3.2 Non-functional requirements

ID	Requirement	Acceptance criterion
NFR1	<i>Reproducibility</i> : the same (scenario, seed) pair must always produce the same event timeline.	The generated timeline is identical for equal seeds and is ordered by tick.

ID	Requirement	Acceptance criterion
NFR2	<i>Testability</i> : it must be possible to drive simulation time by hand and to observe the sequence of events without modifying production code.	The <code>manualClock</code> argument disables the scheduler; an installable probe records events together with their order.
NFR3	<i>Simulated timing</i> : every duration of the world must be expressed in ticks and not in real time, so that behaviour does not depend on machine speed.	Durations of exams, treatments, leases and stays are all in ticks; real time appears only in the deadlines, back-offs and timeouts of the protocols, none of which changes the state of the world.
NFR4	<i>Modularity</i> : the three levels (agents, Java environment, Prolog theories) must be independently replaceable.	The interface between agents and environment is limited to percepts and actions; the only bridge towards Prolog is a single class.
NFR5	<i>Concurrency safety</i> : the model shared between the agent threads and the clock thread must neither become corrupted nor block.	One-way lock hierarchy, concurrent data structures, one Prolog engine per thread.
NFR6	<i>Extensibility</i> : adding a hospital or a doctor must not require changes to the negotiation logic.	Participants are discovered dynamically in the Directory Facilitator, not listed in a configuration.
NFR7	<i>Responsiveness</i> : a doctor must obtain their care plan in a time negligible with respect to the tick.	Plan derivation is in the order of milliseconds for every pathology and code.
NFR8	<i>Build portability and reproducibility</i> : the project must compile and run the tests with no manual installation beyond the JDK.	Gradle wrapper included, CI pipeline on GitHub Actions green on every push.
NFR9	<i>Observability</i> : the course of a negotiation must be readable while the system runs.	The <code>[CNP]</code> , <code>[CNP-NET]</code> , <code>[RENEG]</code> , <code>[REROUTE]</code> , <code>[PREEMPT]</code> trace appears on the MAS console in real time.

Table 4: Non-functional requirements and their acceptance criteria.

3.3 Scope delimitation

Some choices deliberately bound the perimeter of the project. They are conscious simplifications, not omissions: each one is motivated at the point of the report where the corresponding mechanism is described, and taken up again in Section 10.

- **Single-JVM execution**: the Jason infrastructure in use is `Local`. The Directory Facilitator is still employed for the dynamic discovery of participants, but there is no multi-host deployment and no real network protocol is ever crossed.
- **Obstacle-free map**: movement is one greedy Manhattan step per tick, which in

the absence of obstacles is already optimal: there is no path search, and none would be needed.

- **Non-optimal planning:** the planner returns the first valid plan it finds within a length budget, not a provably minimal one.
- **Stylised clinical model:** five pathologies, four severity codes and a fixed catalogue of exams and equipment. The aim is to exercise the coordination mechanisms, not to reproduce a real medical protocol.
- **No persistence and no user interaction:** there are no databases and no commands issued from outside, the graphical view is a read-only observer and a scenario is entirely predetermined at start-up.
- **Static agent population:** agents are declared in the project file and are never created or destroyed at runtime; an inactive patient is simply dormant until the scenario activates it.

3.4 Glossary

Term	Meaning in MedNet
Tick	Discrete time unit of the simulation. Every duration of the world is expressed in ticks.
Severity code	One of red , yellow , green , white , in decreasing order of urgency.
Preliminary triage	Assessment performed on site by the ambulance; it produces an estimate of the code.
Secondary triage	Assessment performed in the hospital by the nurse; it reveals the true code, which may differ from the estimate.
CNP	Contract Net Protocol: the negotiation protocol used at both levels.
CnpId	Identifier of a negotiation: <code>call_N</code> at the network level, <code>t(Patient)</code> at the in-hospital level.
Initiator / Participant	The two roles of the CNP: the one who opens the round and the one who bids.
Walk-in	Emergency external to the network that shows up directly at the door of a hospital and consumes one of its beds.
Divert	Request to reopen the negotiation because the assigned hospital can no longer take the patient.
Preemption	Interruption of a less urgent treatment in order to free a doctor for a red-code case.
Replanning	Fresh derivation of the care plan starting from the exams already performed.

Term	Meaning in MedNet
Lease	Time-limited grant over a resource, beyond which the resource is reclaimed by the system.
Segment	Block of plan that acquires one piece of equipment, uses it and releases it.
Stage	Set of independent segments that can be executed in parallel.
Percept	Literal the environment makes observable to one specific agent.
Situatedness	Property by which an agent perceives only the portion of the world its role and position allow.
DF	Directory Facilitator: the service registry used for dynamic discovery.
Probe	Observation hook on environment events, a no-op in production and a recorder in the tests.
RBAC	Role-Based Access Control: authorisation of actions on the basis of the agent's role.

Table 5: Glossary of domain terms.

4 Course topics applied in MedNet

This chapter makes explicit the main topics of the course applied in the MedNet project. For each concept a short theoretical introduction is given, followed by a reference to its concrete realisation in the system, described in Section 5.

4.1 BDI agents and AgentSpeak(L)

The BDI model describes an agent through three mental attitudes:

- *beliefs*: what the agent holds to be true of the world and of itself;
- *desires*: the states of affairs the agent would like to bring about, which in AgentSpeak(L) take the concrete form of *goals*;
- *intentions*: the courses of action the agent has committed to and is actually executing.

The reasoning cycle is reactive. The agent updates its beliefs from the percepts and the messages it receives, derives events from them and selects, for each event, a plan whose *trigger* unifies and whose *context* is true with respect to the current beliefs. The body of the plan is then executed as an intention.

A plan is not a procedure invoked directly: it is a *conditional* response to an event. The choice of plan therefore depends on the mental state of the agent at the moment the event occurs.

Application in MedNet. In MedNet all the decision logic lives in the BDI level. The beliefs of the agents have three distinct origins:

1. the percepts computed by the environment;
2. the domain knowledge internalised at start-up from the Prolog knowledge base;
3. the beliefs produced by the messages received.

Goals are explicit, for instance `!pickup`, `!treat_patient`, `!cnp_start` and `!process_queue`. Intentions are moreover manipulated explicitly when the domain requires it:

- the doctor executes the segments of a stage as parallel sub-intentions through `!!run_segment`;
- a preemption drops intentions through `.drop_intention` and `.fail_goal`;
- the failure plans `!g` handle cases such as a refusal during a negotiation or the release of a piece of equipment after a segment has been aborted.

The classical properties of agents are observable in the following aspects of the system:

Autonomy A hospital decides on its own whether and how much to bid. The initiator of a negotiation can only formulate a request, without imposing a decision (Section 5.4.2).

Social ability The agents coordinate exclusively through the exchange of messages: there is no shared memory between them (Section 5.4.3).

Reactivity A percept such as `reservation_lost` or the arrival of a `cfp` immediately generates an event. Moreover, the environment forces a fresh perception after every successful action, without waiting for the next tick (Section 5.1.2).

Pro-activeness A nurse does not wait to be prompted in order to drain their own queue, but relauches `!!process_queue` after every event that may have made it servable.

Situatedness What an agent can see and do is determined by its role and its position (Sections 5.4.3 and 5.7.1).

4.2 FIPA ACL communication

The FIPA Agent Communication Language structures communication between agents as an exchange of *communicative acts*. Every message contains:

- a *performative*, which declares the illocutionary force of the act;
- a *content*, that is the term the act refers to.

A performative does not transfer control of the receiving agent. It only conveys information or proposes a request:

- `tell` asserts something into the belief base of the receiver;
- `achieve` proposes the adoption of a goal to the receiver.

In both cases it is the receiver that decides how to react, according to its own plans and its own contexts. This asymmetry distinguishes communication between agents from an ordinary procedure call.

Performatives used. MedNet uses both primitive performatives of Jason:

- `achieve` is used when an agent delegates a goal to another agent, for instance when a patient asks the centre to handle an emergency or when the centre asks an ambulance to perform a pickup;
- `tell` is used to convey information, such as a triage report, an admission confirmation, a divert request or a discharge notification.

The five acts of the Contract Net Protocol,

`cfp`, `propose`, `refuse`, `accept_proposal`, `reject_proposal`,

are carried as the content of `tell` messages and correspond one to one to the FIPA performatives of the same name. The complete vocabulary of the protocol and of the domain is reported in Section 5.4.3.

Persistence of messages. Since a `tell` message leaves a belief in the belief base of the receiver, that belief outlives the act that generated it, and a belief that has outlived its own round is indistinguishable from one that has just arrived. Every consumed message is therefore explicitly abolished; the consequences of failing to do so are discussed in Section 5.4.1.

4.3 Contract Net Protocol: negotiation and coordination

The Contract Net Protocol is a protocol for the distributed allocation of tasks. An *initiator* agent, which has to assign a task but does not know which agent is best suited to carry it out, sends a *call for proposals* to a set of *participants*.

Every participant autonomously evaluates the task on the basis of its own state and answers with:

- a bid, if it is able to carry out the task;
- a refusal, if it cannot take part.

The initiator then collects the answers within a deadline, selects the best bid, accepts it and rejects the others.

The central point of the protocol is that the initiator does not directly hold all the information needed for the decision: it emerges from the negotiation. The Contract Net is therefore particularly well suited when the relevant knowledge is distributed among the candidates, as in the case described in Section 2.2.

Two levels of application. In MedNet the protocol appears at two levels:

1. **Network level:** the control centre assigns the admission of a patient and the hospitals take part as bidders;
2. **Hospital level:** the triage nurse assigns the care of a patient and the doctors of the same hospital take part in the negotiation.

The two levels differ in:

- initiator;
- participants;

- shape of the task;
- service looked up in the Directory Facilitator;
- deadline.

The protocol, however, stays the same. The CNP code is written once and included by both levels; the specific parts are supplied through callbacks (Section 5.4.1). The two-level requirement thus becomes a matter of configuration rather than of code duplication. A fix to the protocol is consequently applied to both negotiations simultaneously.

Handling the dynamic cases. With respect to the classical formulation, the implementation explicitly addresses what the theory leaves implicit and what becomes decisive in a dynamic world: renegotiating a task while a round is still open, late bids, the abortion of a round, and the order between cleanup and callback (Section 5.4.1).

4.4 Knowledge representation and symbolic reasoning in Prolog

The logical representation of knowledge describes the facts and the rules of a domain through Horn clauses, delegating their derivation to an inference engine based on unification and SLD resolution.

With respect to an imperative encoding of the same constraints, this approach separates what is true in the domain from what the program has to do: conclusions need not be enumerated explicitly, but are derived by the inference engine.

Integration in MedNet. MedNet embeds tuProlog as its engine and places four theories in it:

1. clinical and organisational knowledge;
2. access control policy;
3. generic STRIPS planner;
4. care planning domain.

These theories are used by both sides of the system:

- the Java code queries them for job durations, default severity codes and protocol constraints;
- the agents internalise the relevant portion at start-up, so that the context conditions of their plans can read it as an ordinary belief.

The resulting architecture is described in Section 5.1.3.

Derived knowledge. The fact that knowledge is derived, and not merely enumerated, is visible at several points of the system:

- the preemptability of a severity code is defined by a rule: a code is preemptable if its priority is less urgent than the highest one;
- the catalogue of pathologies is derived from the relation between a pathology and the specialisation it requires;
- the scenario configuration reads that derived catalogue, rather than a list encoded separately.

As a consequence, adding a new severity code to the theory updates preemption on its own, with no change to the Java code or to the AgentSpeak agents. The weights of the admission bid and the Java enumeration of the codes are not derived and have to be extended too; this is not left to memory, because the consistency test fails the build until the new code has its weights and the enumeration agrees with the theory again (Section 5.1.4).

4.5 Classical planning with STRIPS

STRIPS-style classical planning represents the state of the world as a set of ground propositions and describes actions through three components:

- *preconditions*: they must be true for the action to be applicable;
- *add list*: propositions made true by the action;
- *delete list*: propositions made false by the action.

Planning means searching, in the state space generated by these actions, for a sequence that leads from the initial state to a state in which the goal condition holds.

The underlying assumption is that the world is deterministic and fully observable from the planner's point of view. Defining precisely the content of the initial state therefore becomes essential.

Care planning. In MedNet the doctor does not know in advance the sequence of exams needed for a given pathology. Instead they specify:

- where the patient is;
- which exams are still pending;
- which exams have already been performed;
- which pieces of equipment are needed;
- which condition identifies the conclusion of the case, namely **discharged**.

The domain and the planner then derive the rest (Section 5.6.1). An important property follows: when a patient is preempted, they can be replanned starting from what has already been done. The new plan is therefore strictly shorter than the initial one (Section 5.6.1).

Design choices. The initial state contains only what the agent may legitimately know: the proposition describing a piece of equipment as available means “free for me to take”, not “globally free”, since the doctor does not perceive the global state of the equipment. Real mutual exclusion is negotiated at runtime.

4.6 Security and access control: RBAC

Role-based access control, or RBAC, decouples permissions from identities:

- permissions are assigned to roles;
- identities are associated with roles;
- authorisation consists in checking that the role of the requester includes the requested permission.

With respect to individually assigned permissions, this approach offers two main advantages:

- the policy stays readable as the number of subjects grows;
- adding a new subject does not require the policy to be reformulated.

A fundamental property is *fail-closed* behaviour: in the absence of a rule that explicitly authorises an operation, access is denied.

Application in MedNet. In MedNet the RBAC policy is declared in Prolog and not in Java, it covers seven roles and seventeen actions, and the environment asks it a single question before executing each action: whether the role of the agent is authorised to perform that type of action. How this single question yields *fail-closed* behaviour for free, and how it combines with the validation of the object of the action, are covered in Section 5.7.1.

5 Design

5.1 Architecture

5.1.1 Overall architecture

MedNet is organised in three levels, shown in Figure 1, each responsible for a different kind of knowledge and written in the language appropriate to that kind.

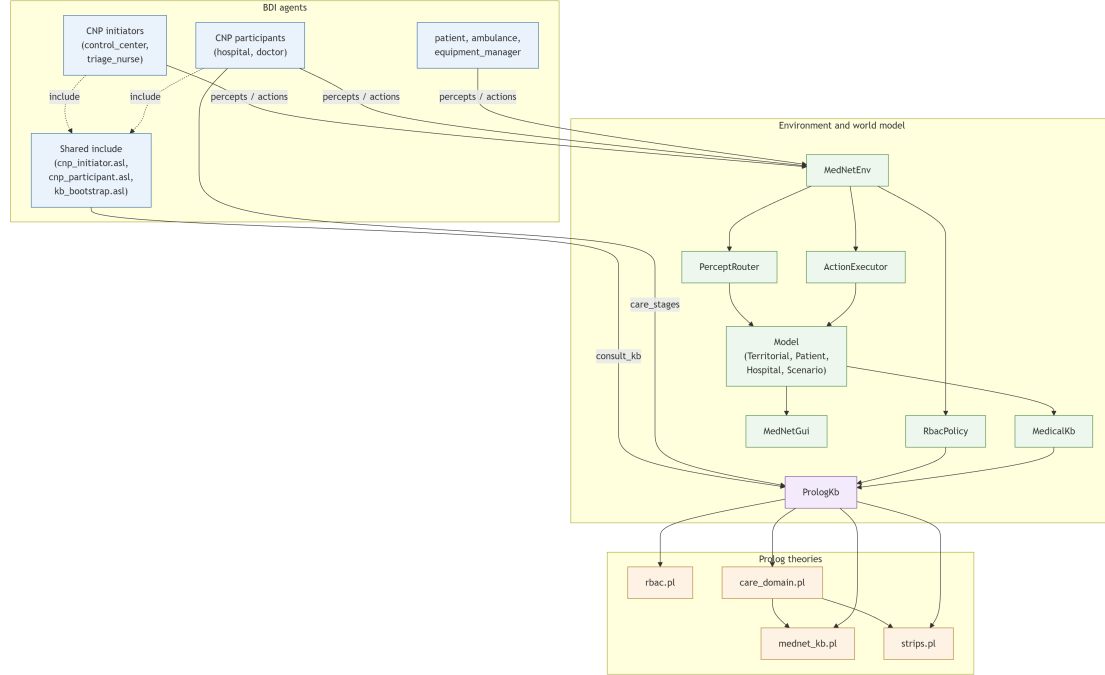


Figure 1: The three-level architecture of MedNet.

BDI agents. Written in AgentSpeak(L) and executed by Jason. All the decision logic lives here: when to bid and how much, when to refuse, when to interrupt a treatment, how to react to a percept or to a message. One file per agent type plus a set of shared code: the negotiation protocol in its two roles and the internalisation of the knowledge base.

Environment and world model. Written in Java. Everything that is *physical* lives here: the territorial grid, the beds, the equipment, the jobs in progress, the discrete clock, the scenarios. This level takes no decisions, there is no Java component that chooses where to send a patient, but it enforces the constraints of the world: an inapplicable action fails, a lock has a single owner, an exam lasts a fixed number of ticks.

Prolog theories. They hold the declarative knowledge: the clinical and organisational

constraints, the access control policy, the STRIPS planner and the care planning domain.

The architecturally relevant point is that the third level is read by both of the other two: Java and AgentSpeak literally query the same clauses, and the consistency between the two sides is checked by the build (Section 5.1.4). The interfaces between the levels are deliberately narrow: the agents communicate with the environment only through percepts and actions, and with Prolog only through two internal actions.

5.1.2 The percept/action cycle (Jason ↔ Java)

The `MedNetEnv` class implements the two methods that make up the whole interface between the BDI level and the simulated world.

In the inbound direction it delegates to `PerceptRouter`, which builds the literals *that specific agent* is entitled to observe, starting from its role and its position. There is no global set of percepts to filter: the list is composed per agent, and this is the concrete implementation of situatedness (Section 5.4.3).

In the outbound direction it first asks the authorisation question and only afterwards delegates to `ActionExecutor`, which validates the arguments and the target and applies the effect to the model (Listing 1). An RBAC denial emits an observable event and returns `false`, which on the agent side shows up as a plan failure. On success the agents re-perceive *immediately* instead of waiting for the next tick, which is what makes the system reactive to its own action without tying reactivity to the period of the clock.

```
1  @Override
2  public boolean executeAction(final String agName, final
    Structure action) {
3      if (!RbacPolicy.check(agName, action.getFunctor())) {
4          ProbeRegistry.current().onEvent("rbac_denied", agName,
              action.getFunctor());
5          LOGGER.warning(() -> "RBAC: denied " + action + " to "
              + agName);
6          return false;
7      }
8      final boolean ok = actions.execute(agName, action);
9      if (ok) {
10         informAgsEnvironmentChanged();
11     }
12     return ok;
13 }
```

Listing 1: The single gateway between agents and world

Both directions are deliberately narrow: an agent can see only what the router emits and can change the world only through the seventeen actions the executor implements. There is no shortcut: no agent holds a reference to an object of the model.

5.1.3 The bridge to Prolog (Java → Prolog, Jason → Prolog)

The `PrologKb` class is the only bridge towards the logic engine. It concatenates the four Prolog files taken from the classpath into a single theory and hands out *one tuProlog engine per thread*: the engine is not thread-safe and the agents reason on their own threads. Since the theories are static and nothing is ever asserted at runtime, the per-thread engines cannot diverge.

On top of this bridge sit two thin facades for the Java side: `MedicalKb`, which exposes durations, default codes and protocol constraints in typed form, and `RbacPolicy`, which exposes the authorisation question alone and memoises its answer.

The agent side reaches the same theory through two Jason *internal actions*, themselves implemented in Java and routed to the same `PrologKb`. The first, `consult_kb(Facts)`, returns as a list of literals the knowledge the BDI level has to reason with; the second, `care_stages(P, Pathology, Code, Done, Stages)`, asks the STRIPS planner for the care plan of a patient starting from the exams already performed. Agents and environment therefore literally query the same clauses.

5.1.4 Why clinical/organisational knowledge lives in Prolog

The clinical theory is the single source of truth for the clinical and organisational constraints, and both halves of the system read it instead of repeating it. Java queries it for the durations of exams and treatments, for the default severity of a pathology, for the catalogue of pathologies and for the constraint that mandates the operating room for red codes. The agents internalise it once at start-up, so that the context conditions of their plans can read it like any other belief.

Internalisation is not a literal copy of the theory, and this is where the choice of Prolog pays off. The predicate that establishes which codes are preemptable is a *rule* (Listing 2), and what the agents receive are the conclusions Prolog derives from it: inference happens in the knowledge base, the answers reach the agents. Likewise the catalogue of pathologies is derived from the relation between pathology and specialisation, and it is in that derived form that the scenario generator reads it.

```
1  % A lower number means more urgent.
2  code_priority(red, 0).
3  code_priority(yellow, 1).
4  code_priority(green, 2).
5  code_priority(white, 3).
6
7  pathology(P) :- requires_specialization(P, _).
8
9  % Red is never preempted; every less urgent code may be.
10 protocol_preemptable(Code) :- code_priority(Code, P), P >= 1.
```

Listing 2: Derived knowledge in the clinical knowledge base.

A second criterion governs what actually gets shipped: only what a plan really reads reaches the belief base. The data about exams, equipment and durations stay in Prolog,

because the doctor derives the care plan with the planner instead of running a hard-wired exam loop, and no AgentSpeak plan mentions those predicates any more. Shipping them anyway would put eighteen literals into the belief base of every agent that no plan can unify with.

The risk of drift between the theory and the code that accompanies it remains, and it is addressed explicitly: the `KbConsistencyTest` class fails the build if the enumeration of severity codes, that of the roles, the admission weights, the equipment list or the job durations stop agreeing with the theory, and `RbacPolicyTest` does the same for the action surface of the access control policy.

5.2 Infrastructure and world model

5.2.1 The discrete clock and listener ordering

The `SimulationClock` class is a global discrete clock. In production a single-threaded daemon scheduler advances it every `period` milliseconds; under the `manualClock` argument the scheduler is not started and ticks are generated programmatically, which is what lets unit tests drive time by hand.

Listeners are notified *in registration order*, and the initialisation of the environment relies on this (Listing 3). The order is not arbitrary: first the territorial world moves, then the internal state of the hospitals advances, then the scenario events of the current tick are generated, then the agents re-perceive, and only at the end does the completion watcher observe the resulting state. The latter is registered last *on purpose*: it is the listener that stops the clock, so the tick that closes the run must still reach all the others in full.

```

1 clock.register(territorial::onTick);
2 hospitals.values().forEach(h -> clock.register(h::onTick));
3 clock.register(generator::onTick);
4 clock.register(tick -> informAgsEnvironmentChanged());
5
6 // Registered last: the watcher stops the clock, so every other
  // listener must
7 // still receive the tick that completes the run.
8 completion = new CompletionWatcher(generator, patients,
  hospitals, clock,
9     this::onSimulationFinished);
10 clock.register(completion);

```

Listing 3: The order of the tick listeners.

Everything that takes time *in the world* is measured in ticks, never in real milliseconds: exam durations, treatment durations, equipment leases, reservation leases, walk-in stays. Real time is used only where what is being measured is *how long to wait for another agent to reason*: the CNP bid collection deadline, the back-offs with which the centre and the nurse retry a negotiation that found nobody, and the timeout on the acquisition of a piece of equipment. None of these is a duration of the simulated world, and none of them moves its state.

5.2.2 The two spatial levels

The world is modelled on two distinct and coupled spatial levels.

The **territorial** level is an obstacle-free 30×30 grid holding the ambulances and the fixed hospital sites. Movement is one cell per tick along the axis with the larger residual distance. In the absence of obstacles this greedy Manhattan step is already optimal, so there is no path search, and introducing one would require replacing a single component.

The **in-hospital** level consists of one model per hospital, holding beds, the priority triage queue, equipment, exam and treatment jobs in progress and triage results per patient. Every instance has its own monitor and every method that modifies its state is **synchronized**, because the agent threads and the clock thread both access it.

The two levels are coupled by the patient, who moves between them: they are on the map while waiting and while being transported, and inside a hospital after being unloaded. The transition from one level to the other coincides with the unload action, which also converts the bed reservation into occupancy: there is no instant in which a patient is in both levels or in neither.

5.2.3 Concurrency handling and lock acquisition order

The simulation is genuinely concurrent: every Jason agent reasons on its own thread, the clock advances on a separate daemon thread, and all of them insist on the same world model. The shared structures are consequently protected, **synchronized** models, **ConcurrentHashMap** registries, **CopyOnWriteArrayList** listener lists, but protecting the individual objects is not enough to rule out deadlock: an order is needed.

The order adopted is a one-way hierarchy. The clock thread acquires, in this order, the monitor of the clock, then that of a hospital, then that of a piece of equipment. The agent threads enter from the action executor and acquire the monitor of the hospital and then that of the equipment, but *never* that of the clock: reading the current tick happens on an **AtomicLong** without any lock.

A third potentially problematic thread is Swing's Event Dispatch Thread. The view is updated exclusively with **invokeLater** and never with **invokeAndWait**, so the EDT can under no circumstances block the clock thread; moreover the view is built asynchronously inside a guarded block, so a window that fails to open costs only the window and not the run.

5.2.4 Simulation termination

The completion watcher declares a run concluded when three conditions hold *simultaneously*:

1. the scenario timeline is exhausted, so no new stimulus can arrive;
2. every patient has been discharged;
3. every hospital is back at full free capacity.

The third condition is what gives meaning to the other two. Without it “the run is over” would only mean that the known patients have been served, whereas with it the phrase means that the world is genuinely *quiescent*. A bed still reserved for a patient admitted elsewhere, or still occupied by a walk-in serving out their stay, keeps the clock running: a lost reservation can therefore not be hidden by the run finishing around it. The condition nevertheless always stays reachable, because the lease on reservations bounds how long an unclaimed bed can stay blocked (Section 5.5.2).

On completion the clock is frozen irreversibly and the world keeps its final state, while the MAS deliberately stays alive: the agents are not killed, so the trace of the negotiation on the console stays readable and the view keeps showing the state reached.

5.3 Modelling

5.3.1 Life of a patient (end-to-end path)

The complete journey of a patient crosses every mechanism of the system, and describing it once in full makes the sections that follow easier to read, each of them going deeper into one stretch of it.

1. **Activation.** At the scheduled tick the scenario generator produces the event that activates a patient. The patient agent starts perceiving that it is active, its own condition and its own position, and sends exactly one request to the control centre. The three activation percepts may arrive in any order, so each of them retries the call, and an atomic guard on an “already reported” belief ensures that only one goes out.
2. **Ambulance dispatch.** The centre assigns an identifier to the call and atomically claims the nearest free ambulance, computed on its own position percepts; it then delegates a pickup goal to it. The claim is atomic and separate from sending the mission, because two calls handled concurrently must never pick the same vehicle.
3. **On-site assessment.** The ambulance reaches the cell, performs the preliminary triage, which generates a percept visible only to *that* ambulance, loads the patient and reports pathology and preliminary code to the centre.
4. **Network negotiation.** The report opens the macro-level round towards the hospitals, querying first the specialists in the pathology and falling back on the generic service only if none is registered. The winner reserves a bed and the destination is communicated to the ambulance (Section 5.4.1).
5. **Transport and delivery.** The ambulance reaches the hospital site and unloads the patient, which converts the reservation into occupancy; it hands the case over to the triage nurse and reports the delivery to the centre, which *definitively closes* the negotiation (Section 5.5.2).

6. **Secondary triage.** The nurse performs the second assessment, which reveals the true code of the patient, inserts them into the priority queue and opens the micro-level round towards the doctors of their own hospital.
7. **Care.** The winning doctor asks the planner for the care plan starting from the exams already performed, and executes it stage after stage: they acquire the equipment through the arbiter, run the exams, the independent ones in parallel, treat the patient and finally discharge them, reporting this to the nurse (Section 5.6.1).

Along this path the patient may be rerouted during transport if the reserved bed is taken away from them (Section 5.5.2), and may be preempted during care if a red-code case arrives while no doctor is free (Section 5.5.1). Figure 2 shows the same path as a sequence diagram.

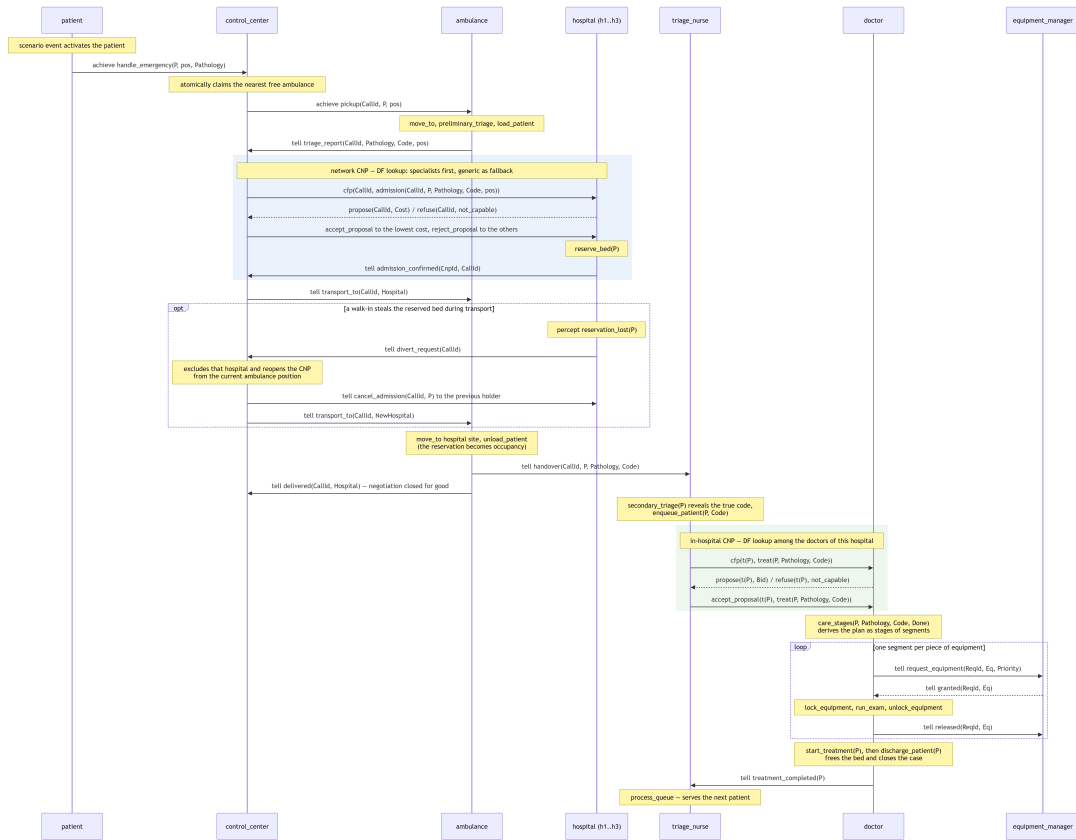


Figure 2: Sequence diagram of the end-to-end path of a patient.

5.4 Interaction

5.4.1 The two-tier Contract Net Protocol

Shared protocol library The protocol is written once, in two files included by the agents that take part in it: `cnp_initiator.asl` and `cnp_participant.asl`. Whoever includes them supplies the domain-specific parts in the form of callbacks, listed in Table 6.

Role	Must supply
Initiator	the belief <code>cnp_deadline(Ms)</code> and the plans <code>!cnp_awarded(CnpId, Winner, Task)</code> and <code>!cnp_no_winner(CnpId, Task)</code> ; it may moreover add <code>excluded(CnpId, Agent)</code> beliefs to exclude candidates.
Participant	the plans <code>!make_bid(CnpId, Task, Bid)</code> and <code>!on_award(CnpId, Task)</code> .

Table 6: The callbacks the two roles of the CNP must supply.

The library takes care of everything else, and in particular of failure handling, which every participant would otherwise have to reimplement. A participant whose bidding plan has a context that fails simply lets the goal fail, and the failure plan of the library translates that failure into an explicit refusal (Listing 4). This is a substantial choice, not a convenience: it means that “I cannot” is expressed as *unavailability of the plan* rather than as a bid value, and the participant leaves the round instead of staying in it at a high price (Section 5.4.2).

```

1 +cnp(CnpId, Task)[source(Initiator)]
2   <- .abolish(cnp(CnpId, Task));
3       !bid_or_refuse(CnpId, Task, Initiator).
4
5 +!bid_or_refuse(CnpId, Task, Initiator)
6   <- !make_bid(CnpId, Task, Bid);
7       .send(Initiator, tell, propose(CnpId, Bid)).
8 -!bid_or_refuse(CnpId, Task, Initiator)
9   <- .send(Initiator, tell, refuse(CnpId, not_capable)).

```

Listing 4: Bid or refuse: the failure of the context becomes a **refuse**.

Network level vs. in-hospital level The two levels of negotiation use the same code and differ only in the parameters reported in Table 7.

	Macro level (network)	Micro level (in-hospital)
Initiator	control_center	triage_nurse_<h>
Participants	hospital_<h>	doctor_<h>_*
CnpId	call_N	t(Patient)
Task	admission(CallId, P, Path, Code, pos)	treat(P, Path, Code)
DF lookup	hospital_<spec>, fallback hospital; hospital alone when the round is reopened	doctor_<h> (both)
Deadline	2000 ms	1500 ms

Table 7: The two Contract Net levels side by side.

The five messages of the protocol correspond one to one to the FIPA performatives of the Contract Net and travel as the content of `tell: cfp/2, propose/2, refuse/2, accept_proposal/2, reject_proposal/1`. The difference between the two deadlines reflects the different expected latency: a network round involves agents that have to consult their own capacity percepts, an in-hospital round involves doctors whose judgement is almost immediate.

Round mechanics and critical details The mechanics of a round are the chain `cnp_start` $\rightarrow$ `cnp_round` $\rightarrow$ `cnp_announce` $\rightarrow$ `cnp_award` $\rightarrow$ `cnp_cleanup`, followed by the callback. Three details, which the classical protocol leaves implicit, turned out to be load-bearing in a world that changes while one negotiates.

One round per CnpId at a time. A round collects bids as beliefs and deletes them when it awards; two rounds with the same identifier would therefore steal each other’s proposals, with the consequence that one awards and the other finds the belief base empty, reports “no winner” and retries, going on awarding hospitals after the patient has already been delivered, and leaving a bed reserved for a patient who will never arrive. The initiator therefore keeps a guard and discards every request to open a second round while the first is in flight. This matters because a single real event reaches the centre *twice*: when a walk-in steals the reserved bed, both the hospital and the ambulance standing at its door ask for the negotiation to be reopened (Section 5.5.2).

The announcement opens a clean slate. The collection deadline is in real time and binds nobody: a slow participant may answer *after* the round has already awarded, and its bid stays in the initiator’s belief base once cleanup has gone by. On reopening the same identifier that stale proposal would be collected together with the new ones, and might win at a price that is no longer valid, or even on behalf of a participant that did not answer at all in this round. This is why the announcement abolishes the proposals and refusals of the previous round *before* sending the `cfp`: the window between collection and cleanup of the previous round cannot be closed, the one before the announcement can.

Cleanup precedes the callback. The “no winner” callback retries by invoking the start of the protocol again; if cleanup ran *after* the callback, that attempt would find the guard still raised, would be discarded as “already in progress” and the patient would starve. The order is therefore visible in Listing 5: award, cleanup, callback.

```

1  +!cnp_award(CnpId, Task)
2    <- .findall(offer(Cost, A), propose(CnpId, Cost)[source(A)],
3             Offers);
4    if (.empty(Offers)) {
5        !cnp_cleanup(CnpId);
6        !cnp_no_winner(CnpId, Task);
7    } else {
8        .min(Offers, offer(Best, Winner));
9        .send(Winner, tell, accept_proposal(CnpId, Task));
10       .findall(L, (propose(CnpId, _)[source(L)] & L \==
11                  Winner), Losers);
12       .send(Losers, tell, reject_proposal(CnpId));
13       !cnp_cleanup(CnpId);
14       !cnp_awarded(CnpId, Winner, Task);
15   }.

```

Listing 5: Award: minimum over the costs, explicit refusal to the losers, cleanup before the callback.

The failure of a round is handled by the library too: a failure plan on `cnp_round` releases the negotiation and invokes the “no winner” callback, so a round that breaks cannot block an identifier forever. The *callbacks*, on the other hand, belong to whoever includes the library and must protect themselves. The triage nurse, for instance, serialises their own CNP with a belief that blocks the whole queue of the hospital: a callback that died leaving it raised would block that queue forever, and this is exactly why the nurse defines their own failure plans on both callbacks (Section 5.5.1).

5.4.2 Bid formulas

Hospital level The bid of a hospital is a cost made of three terms:

$$\begin{aligned}
 Bid = & \underbrace{(|H_x - x| + |H_y - y|) \cdot W_d(Code)}_{\text{travel time}} \\
 & + \underbrace{\frac{(Cap - Free) \cdot 100}{Cap}}_{\text{ward load}} + \underbrace{P(Code)}_{\text{specialisation mismatch}}
 \end{aligned}$$

where the distance weight W_d and the mismatch penalty P are read from the knowledge base as a function of the severity code (Table 8), and the penalty is 0 if the hospital holds the specialisation the pathology requires. Listing 6 reports the corresponding plan.

Code	W_d	P
red	30	2000
yellow	20	800
green	10	300
white	10	300

Table 8: Weights of the admission bid.

```

1 // cost = distance*W(Code) + occupancy% + specialization
  penalty P(Code)
2 +!make_bid(CnpId, admission(_CallId, _Patient, Pathology, Code,
  pos(X, Y)), Bid)
3   : ready & beds_free(B) & B > 0 & my_pos(HX, HY) &
    beds_capacity(Cap)
4     & admission_weights(Code, DistWeight, SpecPenalty)
5   <- if (requires_specialization(Pathology, S) &
    my_specialization(S)) {
6     Penalty = 0;
7   } else {
8     Penalty = SpecPenalty;
9   };
10   Bid = (math.abs(HX - X) + math.abs(HY - Y)) * DistWeight
11         + (Cap - B) * 100 / Cap + Penalty.
12 // No fallback plan: with beds_free(0) the context above fails
    and the library refuses.

```

Listing 6: The admission bid. With no free beds the context fails and the library refuses.

Travel time. It measures how long it will take before the patient reaches definitive care: since an ambulance moves exactly one cell per tick, the Manhattan distance *is* the travel time in ticks, and the weight W_d acts as a conversion factor between travel ticks and percentage points of occupancy.

Ward load. It measures how loaded the ward that would receive the patient already is, and it is deliberately a *percentage* and not a count: with absolute values a two-bed hospital would look emptier than a three-bed one at equal saturation and would win systematically, whereas normalisation makes hospitals of different sizes comparable.

Specialisation mismatch. It expresses the clinical appropriateness of the admission, and is the only one of the three terms to be worth 0 when the hospital holds the specialisation the pathology requires.

It is worth noting that *only* locally perceived quantities appear in the formula: free beds, capacity, position and specialisations are percepts of that one hospital, and the centre receives none of them. The capacity of the network is discovered solely through

bids and refusals, and this is what makes the Contract Net the right protocol here instead of a centralised scheduler.

Finally, the weights come from the knowledge base and not from the plan. The trade-off between timeliness, load and appropriateness is clinical policy, not control logic: the plan declares *how the terms combine*, whereas *how much they weigh* is a fact of the domain, declared once, on which a test enforces the invariant that a less urgent code can never weigh distance or mismatch more than a more urgent one.

Doctor level The bid of a doctor is much simpler: it is 0 if their specialisation matches the pathology and 10 otherwise. Availability, on the other hand, is not a term of the bid but a hard constraint in the context of the plan, itself read from the knowledge base (Listing 7): a doctor who is already busy has a false context, the goal fails and the library refuses on their behalf.

```

1  +!make_bid(t(P), treat(P, Pathology, _), Bid)
2      : ready & my_specialization(MySpec)
3      & protocol_max_parallel_patients(doctor, Max) & .count(
4      & treating(_, _), N) & N < Max
5  <- if (requires_specialization(Pathology, MySpec)) {
6      Bid = 0;
7  } else {
8      Bid = 10;
9  }.

```

Listing 7: The bid of a doctor: availability is a constraint, not a cost.

Why costs rather than utilities; why weights increase with severity

Bids are expressed as *costs* and the award picks the minimum, at both levels. The reason is uniformity: keeping every bid as a cost lets the same award code work for hospitals and doctors without a per-level comparator, and it is one of the conditions that make a single protocol library possible.

An unavailable resource must be *absent* from the round and not merely expensive.

A saturated hospital bidding a very high value could still win the moment all the others refuse, and the reroute scenario would break silently; for the same reason there is no fallback bidding plan, which would fire with zero free beds too and would void the guarantee.

It remains to motivate why the weights grow with urgency. For a critical case the time to access care and being in the right hospital must dominate load balancing, whereas a stable case can be sent a little further to relieve a crowded ward. This is observable behaviour, and it is what the **severity** scenario is built to show: two patients call from cells with the same distance profile (11 cells from **h1**, 12 from **h3**) with pathologies every hospital can treat, while two walk-ins keep **h1** at one free bed out of three. Table 9 reports the figures.

	h1	h3	Winner
green	$11 \cdot 10 + 66.67 = \mathbf{176.67}$	$12 \cdot 10 + 0 = \mathbf{120.00}$	h3
red	$11 \cdot 30 + 66.67 = \mathbf{396.67}$	$12 \cdot 30 + 50 = \mathbf{410.00}$	h1

Table 9: The **severity** scenario: same geometry, different outcomes. The occupancy term of **h3** goes from 0 to 50 because in the meantime it has admitted the green patient.

With a single weight for every code the red case would score 176.67 against the 170 of **h3** and would follow the green one to **h3**. That the two patients end up in *different* hospitals is therefore the observable consequence of the severity term and not an artefact of the geometry, and it is exactly what the corresponding end-to-end test asserts.

5.4.3 Agent–environment interaction contract

Percepts per role (situatedness) Percepts are built per agent by the router, and nothing else is observable. Table 10 reports the complete contract.

Role	Percepts
patient	active, my_condition(Pathology, GuessCode), at(pos(X,Y)), only once activated, and only while on the map
control_center	ambulance_pos(Amb, pos(X,Y)), hospital_info(HospitalAgent, NurseAgent, X, Y), never the bed count
ambulance	at_target(X,Y), carrying(Patient), triage_data(Patient, Pathology, Code) (only after <i>its</i> <i>own</i> preliminary triage), hospital_info/4
hospital	my_hospital(H), my_pos(X,Y), my_specialization(S), beds_free(N), beds_capacity(N), arrived(P), reservation_lost(P), reservation_expired(P)
triage_nurse	my_hospital(H), waiting(P, Code, SinceTick), triage_result(P, Code), patient_pathology(P, Pathology)
doctor	my_hospital(H), my_specialization(S), patient_pathology(P, Pathology), exam_done(P, Exam) and treatment_done(P) (only for their own jobs), exam_completed(P, Exam) (every patient present here, whoever performed the exam)
equipment_manager	my_hospital(H), tick(T), equipment_state(Eq, free locked(Owner)), lease_expired(Eq, Owner)

Table 10: The situatedness contract: what each role may observe.

The consequences of this contract are all checked by dedicated tests. The capacity of

the network is not known in advance to anyone, because every hospital knows only its own beds and the centre knows none at all. The staff of one hospital never observes the patients of another. An ambulance sees the triage data only of the patient it assessed itself.

Two percepts of the hospital deserve an explicit distinction because they have different meanings.

reservation_lost/1 signals that the bed has been taken away by a walk-in: it is a domain event that opens a renegotiation.

reservation_expired/1 signals that the bed is still blocked after the expiry of the lease for a patient who never arrived: it is a safety net stepping in.

Environment actions The seventeen actions of the environment are listed in Table 11. A malformed or inapplicable action returns **false** and the agent’s plan fails, instead of corrupting the model. Argument validation is a genuine trust boundary: the terms come from the agents’ plans, so anything that is not a plain atom (an unbound variable, a structure, an annotated literal) is rejected before it reaches the model.

Action	Role	Effect / failure condition
<code>move_to(X, Y)</code>	ambulance	Sets the target; one cell per tick until arrival
<code>preliminary_triage(P)</code>	ambulance	Fails if it is not on the same cell as the patient
<code>load_patient(P)</code>	ambulance	Fails if it is not co-located or already carrying someone
<code>unload_patient(P)</code>	ambulance	Fails if it is not on a hospital site, is not carrying P, or that hospital no longer has a bed for P
<code>reserve_bed(P)</code>	hospital	Fails if no bed is free
<code>release_bed(P)</code>	hospital	Fails if there was no reservation
<code>secondary_triage(P)</code>	triage_nurse	Fails if the patient is not present; produces the true code
<code>enqueue_patient(P, C)</code>	triage_nurse	Inserts into the priority queue
<code>dequeue_patient(P)</code>	triage_nurse	Removes from the queue
<code>requeue_front(P, C)</code>	triage_nurse	Puts back at the head of its own priority class
<code>lock_equipment(Eq)</code>	doctor	Single owner, reentrant for the owner
<code>unlock_equipment(Eq)</code>	doctor	Owner only
<code>run_exam(P, E, Eq)</code>	doctor	Fails if the doctor does not physically hold Eq
<code>start_treatment(P)</code>	doctor	Fails without triage; a red code requires the operating room

Action	Role	Effect / failure condition
abort_treatment(P)	doctor	Cancels only the jobs started by the caller for that patient
discharge_patient(P)	doctor	Frees the bed and closes the case
force_release(Eq)	equip. manager	Administrative unlock; that the lease has expired is checked by the caller, not here

Table 11: The action surface exposed by the environment.

Messages Every send is an asynchronous FIPA ACL message: a performative plus a content term. The vocabulary of the Contract Net is identical at both levels and carried as the content of `tell: cfp(CnpId, Task)`, `propose(CnpId, Cost)`, `refuse(CnpId, Reason)`, `accept_proposal(CnpId, Task)`, `reject_proposal(CnpId)`. Table 12 reports the domain messages.

Sender → Receiver	Perf.	Content
patient → control_center	achieve	handle_emergency(P, pos(X,Y), Pathology)
control_center → ambulance	achieve	pickup(CallId, P, pos(X,Y))
ambulance → control_center	tell	triage_report(CallId, Pathology, Code, pos)
control_center → ambulance	tell	transport_to(CallId, Hospital), a second one for the same call is a reroute
hospital → control_center	tell	admission_confirmed(CnpId, CallId), divert_request(CallId)
control_center → hospital	tell	cancel_admission(CallId, P), give the bed back
ambulance → triage_nurse	tell	handover(CallId, P, Pathology, Code)
ambulance → control_center	tell	delivered(CallId, H), ambulance_free, pickup_failed(CallId), transport_failed(CallId, H)
triage_nurse → doctor	tell	preempt_order(NewP, NewPathology)
doctor → triage_nurse	tell	requeue(P, Pathology, Code), treatment_completed(P)
doctor → equip. manager	tell	request_equipment(ReqId, Eq, Priority), released(ReqId, Eq)
equip. manager → doctor	tell	granted(ReqId, Eq), grant_revoked(ReqId, Eq)

Table 12: The vocabulary of the domain messages.

A cross-cutting rule governs the life of these beliefs: every belief carrying a message is abolished as soon as it is consumed. As observed in Section 4.2, a belief that has outlived its own round is indistinguishable from one that has just arrived, and would be counted twice.

Directory Facilitator Service registration and lookup happen through Jason’s Directory Facilitator. Every ambulance registers the `ambulance` service; every hospital registers `hospital` and, once per specialisation it holds, `hospital_<spec>`; every nurse registers `triage_nurse_<h>`; every doctor registers `doctor_<h>`.

Discovery is therefore *dynamic*: neither the centre nor a nurse holds a list of participants. The first round of a network negotiation queries the specialised service and falls back on the generic one only if no specialist is registered; a round reopened after a divert goes straight to the generic service, the destination being constrained at that point by the exclusions rather than by the pathology. A doctor added to the project file therefore takes part in the negotiations immediately, with no other change; a new hospital needs its entry in the scenario configuration as well, since position, beds and specialisations are what the environment turns into the percepts the agent reasons on. In neither case does the negotiation logic change. The preference for specialisation is expressed in the bid and not in the lookup, and this is what allows a generalist to cover for a busy specialist (Section 5.4.2).

The equipment arbiter is the only exception: it is not registered in the DF, and the doctors build its name from the convention, since there is exactly one per hospital and its identity is fixed. Finally, every agent that reasons with the knowledge base internalises it *before* registering, so no negotiation can reach an agent whose belief base is still empty.

Probe events The `EnvProbe` interface is an observation hook on the events of the environment. In production a no-op implementation is installed in it, whereas the tests install a recording one through a registry, and this is what allows them to assert on the timeline of the simulation, ordering included, instead of relying on static latches never reset between one run and the next (Section 7.2).

The events cover the whole life cycle: activation of a patient, walk-in, preliminary and secondary triage, loading and arrival, reservation and release of a bed, enqueueing and requeueing, lock, unlock and forced release of a piece of equipment, start of an exam or of a treatment, abortion, discharge, derivation of a care plan, expiry of a reservation, RBAC denial and conclusion of the simulation.

Two of them carry particular weight in verification. The care plan event is emitted every time a doctor has a plan derived, once per patient under normal conditions, *twice* for a preempted patient, the second time with fewer steps, and this is what makes replanning an assertable property and not merely a declared one. The conclusion event is emitted exactly once, and it is the signal on which every end-to-end test synchronises its own final state check.

5.5 Behaviour

5.5.1 Preemption

Preemption is the answer of the system to a critical case that arrives when no doctor is free. It is deliberately placed at a precise point of the protocol: the decision belongs to the triage nurse and is taken *only* from the “no winner” callback, that is only after the ordinary negotiation has genuinely failed to find an available doctor.

The victim chosen is the *least* critical patient currently in treatment among those the protocol allows to be interrupted (Listing 8). Preemptability is not enumerated in the plan but derived from the knowledge base, with the consequence that red is never a victim and that adding a new severity code requires no change anywhere (Section 5.1.4).

```
1  // The least critical patient in treatment whose code the
   // protocol allows to preempt.
2  preemption_victim(D, P, C) :-
3      .findall(victim(Prio, D2, P2, C2),
4              (under_treatment(P2, D2, C2) & protocol_preemptable
5                (C2) & code_priority(C2, Prio)),
6              Victims)
7      & not .empty(Victims) & .max(Victims, victim(_, D, P, C)).
8  // No doctor free and the waiting patient is red, so preempt.
9  +!cnp_no_winner(t(Patient), treat(Patient, Pathology, red))
10     : preemption_victim(Doctor, Victim, VictimCode)
11     <- .send(Doctor, tell, preempt_order(Patient, Pathology));
12         dequeue_patient(Patient);
13         -queued(Patient, _, _);
14         +under_treatment(Patient, Doctor, red);
15         -assigning(Patient);
16         !!process_queue.
```

Listing 8: The preemption decision: the victim is the least critical case.

The doctor’s side is *cooperative*, never a forced requisition, and unfolds in five steps:

1. the intention driving the case in progress is dropped;
2. every segment still running is made to fail: each failure activates the handler of that segment, and it is that handler that *releases the equipment* (Listing 9), so that nothing is ever torn away from under an exam that is physically in progress;
3. the jobs are cancelled in the model;
4. the victim is sent back to the nurse and re-enters the queue through `requeue_front`: the queue assigns a patient an arrival sequence number once and keeps it, so the victim is reinserted at the head of its own priority class rather than at the tail. The guarantee is the queue model’s, and it is pinned there by a test; the nurse, which picks the next case from the `waiting/3` percept, reads in it the tick of the

last enqueue, so a victim requeued late may still find ahead of it a patient of the same class who has been waiting longer;

5. the red case starts.

```
1 // An aborted segment must always release what it holds and
  // still hit the barrier.
2 -!run_segment(P, Segment)
3   : .member(acquire(Equipment), Segment) & holding(Equipment,
4     P, Equipment)
5   <- .print("segment of ", P, " aborted; releasing ",
6     Equipment);
7     !release(Equipment, P, Equipment);
8     -my_segment(P, Segment);
9     !stage_barrier(P).
```

Listing 9: The failure handler of a segment: it releases what it holds and still reaches the barrier.

Two further branches handle the races in which the preempt order arrives late: if the doctor has finished in the meantime, they simply take the red patient; if they are already engaged on a red case, which is not preemptable, they return the new patient to the queue instead of dropping one of the two.

The most interesting property is that **the victim is replanned, not restarted from scratch**. The completed exams outlive the cancellation of the jobs, and the router makes them observable as a durable percept for every patient present in the hospital, whoever performed them; when the case resumes, the planner receives those exams as already done and returns a strictly shorter plan (Section 5.6.1).

Finally, aborting the jobs still goes through the environment, where RBAC authorisation covers the *role* but not the *object*: validation requires the patient to be present in that hospital and limits the cancellation to the jobs started by the calling doctor, so that a colleague cannot wipe out someone else's case (Section 5.7.1). A cancellation that hits nothing is still a success, because a victim preempted between two stages may have no job in progress and the preemption protocol must not break because of that.

5.5.2 Renegotiation during transport

A *walk-in* scenario event models an emergency external to the network that shows up at the door of a hospital. If no bed is free, the event *takes away a reservation* and the model returns the name of the displaced patient; the hospital then perceives the loss and asks the centre for a divert.

The centre excludes that hospital from the next round and reopens the negotiation starting from the *current* position of the ambulance, so that the new decision accounts for how far the transport has already got. The ambulance receives a second transport order for the same call, drops the intention in progress and reroutes with the patient still on board.

Two safety properties are enforced explicitly.

A reroute gives the previous bed back. The award callback, if it detects that the call was already assigned elsewhere, sends a cancellation to the previous holder *before* rerouting the ambulance.

Delivery closes the negotiation. From that moment on divert requests, transport failures and “no winner” retries are all no-ops, and an award still in flight is cancelled.

The lease on reservations is only the *safety net* for a bed held for a patient who never arrives. With the protocol working it never fires, and the end-to-end tests assert that it does not.

The peripheral failure paths matter too, because that is where a lost event turns into a stall. When a delivery fails, the ambulance must *stay on mission* if it is still carrying the patient, waiting for a new destination, and declare itself free only if it is empty: these are two distinct failure plans, told apart by whether or not somebody is on board. Symmetrically, a handover that arrives before the nurse is on duty is not discarded but waits for the on-duty percept and retries, because a lost event here would leave the patient in a bed nobody will ever examine, and the run could no longer reach the quiescent state.

5.5.3 Mutual exclusion on shared equipment

Diagnostic equipment consists of physical resources contended between the doctors of the same hospital, and the system protects them at two levels, both necessary.

The **physical level** is a single-owner monitor. The lock is reentrant *for the owner*, a retry after a partial failure must not deadlock against one’s own lock, and it is denied to anybody else; only the owner can release, and the forced release is the administrative way out.

The **cognitive level** is one arbiter agent per hospital, which keeps a queue of requests and grants the equipment to the minimum over (priority, request tick): a red-code request jumps the queue, but a piece of equipment already physically busy is never torn away. The granting plan is atomic (Listing 10), because choosing the winner and recording the grant must not interleave with another request: otherwise the same free machine would be granted to two doctors.

```
1 // Picking the winner and recording the grant must not
   // interleave with another request,
2 // or the same free machine is handed to two doctors.
3 @grant[atomic]
4 +!try_grant(Equipment)
5     :   equipment_state(Equipment, free) & not grant_out(
6         Equipment, _, _)
7         & .findall(w(Priority, Since, ReqId, Doctor),
8                   waiting_req(Equipment, req(ReqId, Doctor,
9                                     Priority, Since)),
10                  Requests)
11         & not .empty(Requests)
12     <- .min(Requests, w(_, _, BestReq, BestDoctor));
13     -waiting_req(Equipment, req(BestReq, BestDoctor, _, _));
```

```

12      ?tick(Now);
13      +grant_out(Equipment, BestReq, BestDoctor);
14      +grant_tick(Equipment, Now);
15      .send(BestDoctor, tell, granted(BestReq, Equipment)).
16 +!try_grant(_) <- true.    // busy, already granted, or nobody
                             waiting

```

Listing 10: The atomic granting of a piece of equipment by the arbiter.

The two levels meet in the execution of the exam, which starts only if the requesting agent *physically holds* the lock (Listing 11). A cognitive grant alone is not enough: that check is the point where mutual exclusion is actually enforced, and it is what makes the arbiter a *contention optimisation* and not a trusted authority.

```

1  // The equipment manager's grant is not enough: the exam starts
   only if this
2  // doctor holds the lock. Mutual exclusion over the machines is
   enforced here.
3  final boolean ownsLock = hospital.equipment(equipmentName)
4      .map(eq -> agent.equals(eq.state().owner()))
5      .orElse(false);
6  if (!ownsLock) {
7      return false;
8  }

```

Listing 11: The point where mutual exclusion is actually enforced.

No segment of a care plan ever holds two pieces of equipment: each one acquires exactly one, uses it and releases it, without ever waiting for a second while holding the first. It is the planning domain that gives the plan this shape, and a test asserts it on the linear plan of every pathology. A doctor may well hold two machines at the same time, when two independent segments of the same stage run in parallel, but since neither of the two sub-intentions waits on the other there is no hold-and-wait, and therefore no cycle of waits: the absence of deadlock follows from the shape of the segments, by construction and not by convention.

Two watchdogs recover the stuck states.

Grant expiry. It revokes a grant not claimed within ten ticks, but only if the equipment still turns out to be free: this guard is what makes the revocation safe, because a grant already claimed is never revoked from under an exam in progress. The same watchdog is also what drains the request of a doctor whose segment was aborted while it was still queued, since no message tells the arbiter to cancel a request.

Lease expiry. It releases by authority a lock held beyond expiry. It is the counterpart, on equipment, of the lease on reservations (Section 5.5.2).

One last detail concerns the request identifier, which is deterministic: two attempts for the same patient–equipment pair share the same identifier. If an acquisition attempt is aborted while suspended waiting, a grant arriving right afterwards would stay in the

doctor's belief base and, on the replan of the same patient, the wait would unblock on that leftover without the arbiter having granted anything. This is why the acquisition plan abolishes its own grant *before* sending the request: the physical lock would have held anyway, but the priority ordering of the arbiter would have been bypassed.

5.6 Data and consistency

5.6.1 Planning: Prolog + STRIPS

The planner and the care domain The planner is domain-independent: it knows only the action schemas in the form `action(Name, Pre, Add, Del)` and the relation `holds_in(Condition, State)`, both supplied by the domain theory. The search is a length-bounded forward depth-first search, with a check on already visited states that prunes the acquire-and-release cycles the domain admits (Listing 12). Since states are unordered sets of ground fluents, equality between states is mutual inclusion.

```

1 plan_from(State, Goal, _, _, Acc, Acc) :-
2     conditions_met(Goal, State).
3 plan_from(State, Goal, Budget, Seen, Acc, Plan) :-
4     Budget > 0,
5     Left is Budget - 1,
6     action(Action, Preconditions, Add, Delete),
7     conditions_met(Preconditions, State),
8     apply_effects(State, Add, Delete, Next),
9     \+ visited(Next, Seen),
10    plan_from(Next, Goal, Left, [Next | Seen], [Action | Acc],
11             Plan).
```

Listing 12: The heart of the search: applicability, effects, visited states, budget.

The domain describes the care of a patient by a doctor through five action schemas: acquire a piece of equipment, run an exam, release it, treat and discharge. They operate over fluents representing the pending and completed exams, the available and held equipment, the fact of having been treated and the fact of having been discharged (Listing 13).

```

1 action(acquire(Equipment),
2        [available(Equipment), needed(Equipment)],
3        [held(Equipment)],
4        [available(Equipment)]).
5
6 action(exam(Exam, Equipment),
7        [pending_exam(Exam), equipment_for(Exam, Equipment),
8         held(Equipment)],
9        [done(Exam)],
10       [pending_exam(Exam)]).
11
12 % The case is closed with empty hands: everything acquired has
13    been released.
14
15 action(discharge,
16        [treated, nothing_held],
```

```

14         [discharged],
15     [] ) .

```

Listing 13: Three of the five action schemas, with preconditions, add list and delete list.

A doctor, therefore, knows no sequence of steps for a pathology: they declare where the patient is and what counts as a concluded case, and receive the rest. It is worth noting that the discharge condition explicitly requires that nothing is held any more, which rules out by construction the plans that close a case leaving a piece of equipment locked.

The linear plan thus obtained is then grouped into **stages of independent segments**: a segment is a block that acquires, uses and releases a single piece of equipment, and consecutive exam segments on *different* pieces of equipment are grouped into a single stage, whereas treatment and discharge stay stages of their own. The doctor executes the segments of a stage as parallel sub-intentions and the stages in order, synchronising on a counter barrier whose clauses are atomic, being a read-modify-write shared between concurrent sub-intentions. The parallel diagnostic phase is therefore *derived* from the constraints on the equipment, not hard-wired.

The vocabulary of the plan steps corresponds one to one to the actions of the environment: acquiring means doing the handshake with the arbiter and then taking the lock, running an exam means starting it and waiting for the completion percept, releasing means unlocking and notifying the arbiter, treating and discharging correspond to the actions of the same name. All the waits are on a plain literal and not on an event trigger: Jason evaluates the formula against the belief base on entry and re-evaluates it on every subsequent event, so a percept arriving *before* the plan reaches the wait is not lost.

Search heuristic vs. optimal search The planner returns the *first* plan it finds, not a provably minimal one. This is an intentional trade-off: a search with an admissible heuristic or an iterative deepening would cost seconds here for every red case, whereas a doctor has to answer in milliseconds, and in exchange for that cost the plans obtained are the same. Knowledge was therefore placed in the domain rather than in the search, and a test pins the resulting plan for every pathology, so that a change to the domain surfaces as a failing test and not as a silently different plan at runtime.

Two mechanisms, both in the domain, are what turns a blind search into an immediate one.

The order of the clauses, which is in fact the heuristic: exam before release, before treatment, before discharge, before acquisition. From it follows the strategy “make progress on the patient, then free the equipment, and look for a new one only when nothing else is applicable”, from which emerges the interleaved plan a hospital actually wants, instead of one that grabs all the equipment at the start and holds on to it.

The necessity precondition on acquisition, by which a doctor never looks for a piece of equipment the current case does not need; the cut in the corresponding clause

makes the check semi-deterministic, so the planner does not see the same successor state twice.

Replanning and already-completed stages Replanning rests on a distinction between two percepts that would be easy to confuse.

`exam_done/2` is the completion signal of a job *started by this doctor*, and it disappears when the job is cancelled: it serves to unblock the wait inside a segment.

`exam_completed/2` is instead the *durable* trace that an exam has been performed on a patient present in that hospital, whoever performed it, and it outlives the cancellation of the jobs.

When a preempted case resumes, the doctor collects all the durably completed exams for that patient and passes them to the planner as already done; the initial state no longer holds them among the pending ones, nor does it hold the equipment that served only them. The plan returned is therefore strictly shorter, and verifiably so: the end-to-end test on preemption asserts exactly this, observing two plan derivations for the same victim, the second with fewer steps.

That this works without any code dedicated to “resuming” is the concrete pay-off of the choice to derive plans instead of writing them. There is no resumption procedure: there is a planner to which a different initial state is described.

5.7 Security

5.7.1 RBAC

The access control policy is **not** written in Java. The RBAC theory declares the roles, the way the name of an agent maps onto a role and which actions each role may perform; the Java side is a thin facade that asks the theory a single question for every action received by the environment, before the executor sees it (Listing 1). The decisions are memoised¹, because the question is on the hot path while the theory is immutable.

The logical formulation makes the behaviour *fail-closed on both axes, and for free* (Listing 14): an action no clause names, or an agent whose name matches no role, simply has no proof, and no proof means denied. There is no **else** branch to remember to write. Every denial emits an observable event and is recorded in the log; the end-to-end tests assert that none ever occurs, because a denial in a normal run means that an agent attempted an action its role does not provide for, that is a design error in the plans.

```
1 permitted(doctor, lock_equipment).
2 permitted(doctor, unlock_equipment).
3 permitted(doctor, run_exam).
4 permitted(doctor, start_treatment).
5 permitted(doctor, abort_treatment).
6 permitted(doctor, discharge_patient).
```

¹Stored (cached) so it can be quickly retrieved later without repeating the same calculation.

```

7
8 permitted(equipment_manager, force_release).
9
10 % The single question the environment asks before executing any
   action.
11 can(Agent, Action) :- role_of(Agent, Role), permitted(Role,
   Action).

```

Listing 14: The policy: permissions per role and the single question the environment asks.

The complete matrix is 7 roles $\times$ 17 actions and is pinned exhaustively by a test; it is summarised in Table 13. The last two rows are the most interesting: the two agents that drive the whole negotiation have *no* power to change the physical world, and everything they obtain they obtain by convincing somebody else.

Role	Permitted actions
ambulance	move_to, preliminary_triage, load_patient, unload_patient
hospital	reserve_bed, release_bed
triage_nurse	secondary_triage, enqueue_patient, dequeue_patient, requeue_front
doctor	lock_equipment, unlock_equipment, run_exam, start_treatment, abort_treatment, discharge_patient
equipment_manager	force_release
control_center	no action (acts only through messages)
patient	no action (acts only through messages)

Table 13: The RBAC matrix.

The most important design point is that **RBAC authorises the role, not the object**. The matrix establishes that a doctor may run an exam or abort a treatment; *on whom* it is lawful to do so is a decision that depends on the state of the model, and is therefore taken in the executor, where every action validates its own target: the patient must be present in that hospital, the equipment must be owned by the caller, the cancellable jobs are only one’s own (Sections 5.5.1 and 5.5.3). The two halves together, and neither of them alone, form access control: the first answers “can this role perform this action?”, which is a universal and static question; the second answers “on this object, right now?”, which is a local and dynamic one.

Finally, the naming convention for agents is load-bearing, because both the role and the placement are derived from it. Resolution proceeds from the longest prefixes to the shortest ones, so that a name such as `triage_nurse_h1` never resolves to a shorter role; the theory mirrors the same order in its own clauses, where it is equally load-bearing because resolution cuts on the first match. A test checks that the two sides declare the

same roles and that every name resolves to the role expected of it; keeping the two orders in step, on the other hand, remains a convention.

6 Implementation and technologies

6.1 Technology stack and repository structure

The technology stack is reported in Table 14. All versions are declared in a version catalog, so that updating a library is an operation local to a single file.

Component	Version	Role
Java	21	Language of the environment and of the world model
Gradle	9.7.1	Build, MAS execution, separation of the test tasks
Jason	3.3.0	AgentSpeak(L) interpreter, messaging infrastructure, DF
tuProlog	4.1.1	Embedded inference engine for the four theories
JUnit	6.1.3	Test framework
AssertJ	3.27.7	Fluent assertions

Table 14: The technology stack.

The structure of the repository mirrors the three levels of the architecture:

- **Agents:** in `src/main/asl`, with the shared parts in `include/`.
- **Java code:** in `src/main/java/mednet`, split into `env` (environment, percept router, action executor, probe), `model` (clock, hospital, patient, territory, scenario), `prolog` and `plan` (bridge and internal actions), `rbac` and `view`.
- **Prolog theories:** in `src/main/prolog`.

The organisation of the project is completed by:

- **Tests:** in `src/test`, articulated into Java code, mock agents (`src/test/asl`) and MAS test projects (`src/test/mas2j`).
- **Build resources:** both `src/main/asl` and `src/main/prolog` are included among the resources, so theories and agent files end up on the classpath and can be loaded uniformly by the tests as well.

Communication protocol. The agents communicate exclusively through Jason’s ACL message passing, which on the `Local` infrastructure is realised inside a single JVM: no real network protocol is ever crossed. The choice is justified by the goals of the project, which concern coordination logic and not transport; what matters, and what has been preserved, is that the agents do *not* share memory and that all coordination goes through asynchronous messages. What a multi-host deployment would entail is discussed in Section 10.

Representation of in-transit data. The messages carry AgentSpeak logic terms, not serialised objects: a bid is a term `propose(CnpId, Cost)`, a task is `admission(CallId, Patient, Pathology, Code, pos(X,Y))`. The same shape holds for the percepts, which are literals built by the environment and parsed uniformly, and for the arguments of the actions, which the executor validates as plain atoms before touching the model (Section 5.4.3). The advantage is homogeneity: everything that crosses a boundary of the system is a term, hence unifiable in a plan context without conversions.

Querying the knowledge base. The bridge towards tuProlog exposes a small set of primitives, checking whether a formula is provable, obtaining the first solution or all the solutions, extracting an integer or an atom, and everything else rests on them (Section 5.1.3).

Authentication and authorisation of the components. There is no cryptographic authentication: the identity of an agent is its name, guaranteed unique by the Jason runtime. Authorisation, on the other hand, is systematic and goes through the single point described in Section 5.7.1.

6.2 Build, CI/CD, conventional commits

The build scans the root of the project for `*.mas2j` files and registers an execution task for each of them: adding a launch configuration requires no change to the build. The tests are separated into two tasks, told apart by a tag: the ordinary one excludes the MAS tests, the integration one includes them and assigns one JVM per class (Section 7.2); a third task runs both.

The continuous integration pipeline, on GitHub Actions, runs the whole verification on every push to the development branch, on Temurin 21, and uploads the reports on failure. A second job checks the commit messages.

Commit messages indeed follow the **Conventional Commits**² convention, enforced by commitlint through a Husky³ hook on `commit-msg`: the check happens, that is, while the message is being written, when fixing it costs nothing, rather than at publication time, when it would require rewriting history.

²<https://www.conventionalcommits.org/en/v1.0.0/>

³<https://typicode.github.io/husky/>

7 Testing and validation

7.1 Unit, protocol, and end-to-end levels

The suite comprises many tests, spread over three deliberately separated levels. The separation is not organisational but methodological: each level isolates a different class of errors, and a test placed at the wrong level would either not fail when it should, or fail for reasons that have nothing to do with what it intends to check.

7.1.1 Unit level

No MAS involved. The world model, the theories, the planner and the RBAC matrix are checked directly. Some classes drive the environment itself under `manualClock`, so that every tick is explicit and the assertions are exact instead of timing-dependent: the action surface with its failures (FR3, FR16, FR18), the situatedness contract role by role, including what each role must *not* see (FR5, FR19), and the completion conditions (FR21). The others cover bed accounting and expiries (FR20), the ordering of the triage queue (FR11), mutual exclusion (FR16), movement, the determinism of the scenarios (NFR1), the planner with its invariants (FR15, FR17) and the consistency between Java and theory (NFR4).

Command to run: `./gradlew test`.

7.1.2 Protocol level

A real Jason MAS, but with *mock* agents and a minimal environment with no world model and no RBAC. The real agent under test, the control centre or the nurse, talks to mocks that bid fixed values and expose their own outcomes through a pseudo-action recorded as an event. This isolates the negotiation logic from the physical simulation: the expected winner of every round is determined by the bid table of the mocks, not by the geometry of the map, and a failing test points at a problem in the protocol and not in the model. The tests cover the network award (FR4, FR7), the in-hospital one (FR12), preemption (FR13), renegotiation (FR8) and the double-trigger case (FR10).

Command to run: `./gradlew masTest`.

7.1.3 End-to-end level

The complete set of agents on a deterministic scenario. These tests assert three things: *liveness*, that is that every patient is eventually discharged; *ordering*, that is that the life cycle of each patient is monotone in its phases; and the *absence* of the safety-net events. This last point is the most characteristic of the approach adopted: an RBAC denial, a forced release of equipment or the expiry of a reservation are not tolerated, because each of them means that the protocol has failed and that the environment had to make up for it. A run that ends well *thanks to* a safety net is a run with a bug, and the tests treat it as such.

Command to run: `./gradlew masTest`.

7.2 Practical constraints

Three practical constraints shaped the setup, and they are worth stating because each of them arose from an observed failure and not from a theoretical precaution.

The first is that the MAS tests require **one JVM fork per class**. The Jason runtime keeps global state, so two MAS started in the same JVM interfere; the `forkEvery = 1` configuration on the integration task gives every class its own JVM.

The second is **headless execution with no graphical server**. The view is built only if the dedicated argument is present *and* the JVM is not headless, and the test launcher passes Jason a logging configuration that prevents it from opening its own Swing console. The practical consequence is that CI does not need `xvfb`.

The third is the use of a **resettable probe instead of static latches**. A recording probe is installed before the run and restored at the end, one per class; this provides observability without global state surviving between one run and the next, and above all it enables assertions on the *order* of the events, which latches could not express.

7.3 What the tests do not cover, and why

One gap is deliberate and must be stated: the `default` scenario, the demo one, has no end-to-end test of its own. Its timeline is exercised at the unit level, where the determinism of the seed, the ordering of the events by tick and the effect of a walk-in on the hospital model are all asserted, but no test ever runs the complete set of agents on it. It is the scenario with five pseudo-random patients generated from a seed, and the four deterministic end-to-end scenarios already cover every branch of the protocol: complete life cycle, preemption with replanning, reroute during transport and severity-aware routing. Automating `default` as well would add execution volume and one more JVM fork without adding any verified property.

This scenario therefore constitutes the **acceptance test** of the project, and is verified manually in a reproducible way.

8 Deployment and execution

8.1 Main scenarios

A scenario is a static description made of a hospital network, a fleet of ambulances and a timeline of events, ordered by tick and entirely precomputed at start-up: the same (scenario, seed) pair therefore always produces the same run. The events are of two kinds only, the activation of a patient and the arrival of a walk-in. No agent sees this object: the capacity of the network stays discoverable only through percepts and negotiation.

Scenario	What it demonstrates
default	The demo: 5 pseudo-random patients from a seed, plus a burst of walk-ins that saturates <code>h1</code> during transport.
e2e	Deterministic run with 4 patients, mixed pathologies, no walk-ins: the complete life cycle.
preemption	A yellow case is in treatment on the only available doctor when a red one arrives.
divert	A red case is routed to <code>h1</code> when three walk-ins steal its bed during transport.
severity	Two patients with identical geometry and different codes, to expose the severity term of the bid (Table 9).
empty	Network and fleet with no events at all: used by the unit tests as a fixture.

Table 15: The available scenarios and the phenomenon each is built to show.

It is worth noting how the scenarios are built to *isolate* a phenomenon: in **severity** the two patients have the same distance profile and pathologies every hospital can treat, so that the only remaining variable is the severity code; in **preemption** the corresponding MAS project declares a single hospital with a single doctor, so that the ordinary negotiation cannot but fail.

8.2 Build and run instructions

The project is based on Gradle, with Java 21, and the Gradle wrapper is already included in the repository, so no manual installation is needed. The commands are reported in Listing 15.

```
1  # Runs the simulation with the GUI
2  ./gradlew runMednet_localMas
3
4  # Unit tests
5  ./gradlew test
6
7  # MAS integration tests
8  ./gradlew masTest
```

```

9
10 # Both kinds of test
11 ./gradlew check

```

Listing 15: Building, running the demo and verifying.

The environment is configured through the arguments on the `environment:` line of a `mas2j` file, listed in Table 16. Since the build registers an execution task for every `*.mas2j` file present in the root, experimenting with a different configuration only requires adding a file.

Argument	Effect
<code>seed=<long></code>	Seeds the scenario generator; the same (scenario, seed) pair always produces the same timeline.
<code>scenario=<name></code>	One of those in Table 15.
<code>period=<ms></code>	Real milliseconds per simulation tick (default 250).
<code>gui</code>	Opens the dual view; ignored in a headless JVM.
<code>manualClock</code>	Does not start the scheduler: ticks must be generated programmatically. Used by the unit tests for full determinism.

Table 16: The environment arguments.

What to expect. At start-up the MAS console shows the registration of the agents in the Directory Facilitator, then the trace of the simulation in real time: the emergency calls, the ambulance dispatches, the negotiation rounds at both levels, any reroutes and preemptions, the derived care plans and the discharges. The prefixes `[CNP]`, `[CNP-NET]`, `[RENEG]`, `[REROUTE]` and `[PREEMPT]` make each phenomenon easy to spot. The graphical window, if open, puts a panel per hospital with beds, doctors, triage queue and equipment locks next to the territorial map. The run terminates on its own and the status bar reports the final tick.

9 Self-evaluation

9.1 Pier Costante Babini

Role in the group. The work was split by area, but neither of us confined ourselves to a single language: we both wrote Java code, AgentSpeak agents, Prolog theories and tests. I initialised the project, the Gradle build with the automatic generation of the execution tasks and the continuous integration pipeline. On the Java side I took care of the Jason environment and the percept/action cycle, the discrete clock with the ordered chain of listeners, the territorial model, the scenario generator with the completion watcher, the bridge towards tuProlog and the access control facade. On the agent side I wrote those of territorial coordination, patient, ambulance and control centre, together with the two shared Contract Net files and the internalisation of the knowledge base. On the Prolog side I wrote the care planning domain and the clinical knowledge base, and the internal action that exposes the planner to the agents. On the verification front I built the end-to-end tests, those on the environment and on RBAC, the runner that boots a MAS inside a test and the MAS test configurations.

9.2 Marco Morelli

Role in the group. Like my colleague I crossed all three levels of the system. On the Java side I took care of the in-hospital model, beds, priority triage queue, equipment and jobs, of the patient model and of the dual-panel Swing view; I also looked after the update of the Gradle wrapper, of the dependencies and of the GitHub Actions workflow. On the agent side I wrote the in-hospital ones, hospital, triage nurse, doctor and equipment arbiter, and hence the preemption mechanism with the cooperative cancellation of the segments and the two watchdogs of the arbiter. On the Prolog side I wrote the STRIPS planner and the access control policy. On the verification front I built the protocol test level with the mock agents, the minimal supporting environment and the event observation probe, besides the unit tests on the model and on the consistency between theory and Java code. Finally, I took care of the technical documentation and of the writing of this report.

10 Conclusion and future work

MedNet realises a medical emergency network as a two-tier multi-agent system, in which the allocation of patients to hospitals and that of patients to doctors are both solved by negotiation, with the same protocol, by agents that do not know the capacity of the network in advance and discover it by bidding and refusing. Around this core the system integrates the main techniques covered in the course: BDI agents in AgentSpeak(L) situated in a simulated Java environment, FIPA ACL communication and logical knowledge representation in Prolog.

The behaviours demonstrated go beyond basic allocation. A red-code case that finds no free doctors cooperatively preempts the least urgent treatment, whose victim re-enters the queue without losing their place and is then *replanned* preserving the exams already performed. An ambulance whose reserved bed is taken away is rerouted during transport, with no double reservations and without the event, which reaches the centre by two different routes, multiplying the negotiation rounds. Mutually exclusive equipment is coordinated by a priority arbiter on top of a physical lock that remains the only effective authority, and the very shape of the care plans rules out deadlock by construction. All of this is checked by automated tests on three levels which, besides the expected properties, assert the *absence* of any intervention by the safety nets.

A **multi-host deployment** would replace the local Jason infrastructure with a distributed one: the logic of the agents would be unaffected, discovery being already dynamic, but it would require dealing with message loss and reordering and tuning the deadlines to the real latency. **Obstacle-aware movement** would require replacing the movement engine and, along with it, revisiting the interpretation of the distance weight in the bid, which today rests on the identity between Manhattan distance and travel time. An **optimal planner** would be interesting if accompanied by an experimental comparison between plan quality and derivation time, which is precisely the trade-off resolved today in favour of responsiveness. Finally, **richer walk-in dynamics**, with arrivals driven by a distribution rather than by a fixed timeline, would make it possible to study the behaviour of the network under variable load, and with it the robustness of the bid formulas beyond the scenarios built to isolate a single term of them.