Robotics and Autonomous Systems

Lecture 22: Communication in Jason

Richard Williams

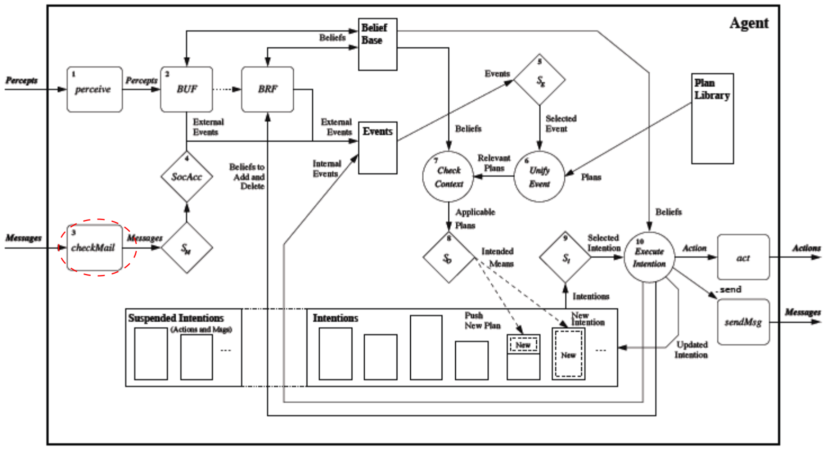
Department of Computer Science
University of Liverpool

UNIVERSITY OF LIVERPOOL

Today

- We will look at communication in Jason.
- This is important since you will have to write agents that communicate as part of the second assignment.
- We will look at general aspects of communication.
- We will then look at a specific example, that of the contract net.

Recall



Messages in Jason

- Each message received by the checkMail method (receiver's perspective) should be thought has having the form:

  `<sender, illoc_force, content>`
- Where:
  - `sender` is the AgentSpeak term with which the agent is identified in the system
  - `illoc_force` is the performative of the message, representing the goal the sender intends to achieve by sending the message
  - `content` is an AgentSpeak formula (varying depending on the performative)

Messages in Jason

- Messages are passed through the use of internal actions that are pre-defined in Jason
- The most typical:
    - `.send(receiver, illoc_force, content)`
    - `.broadcast(illoc_force, content)`

  where `receiver`, `illoc_force` and `content` are as above
- The receiver could also be a list of agent terms
- The `.broadcast` action sends the message to all agents registered in the system

Messages in Jason

- The `.send` and `.broadcast` actions generate messages of the type

  `<sender, illoc_force, content>`

  which are obtained by the checkMail method of **r** (the receiver)
- These messages (recall the previous lecture) are “filtered” during the deliberation cycle of **r** by the SocAcc function which can possibly discard them
  (e.g., because of the type of sender)

Socially acceptable messages

Messages in Jason

- If the message goes through, Jason will interpret it according to precise semantics
    - essentially by generating new events pertaining to the goal and belief bases.

  and **r** might then react to these events according to its plan base

Performatives in Jason

- `tell` and `untell`
  s intends **r** (not) to believe the literal in the content to be true and that **s** believes it
- `achieve` and `unachieve`
  s requests **r** (not) to try and achieve a state-of-affairs where the content of the message is true
- `askOne` and `askAll`
  s wants to know whether **r** knows (anybody knows) whether the content is true.

Performatives in Jason

- `tellHow` and `untellHow`
  s requests **r** (not) to consider a plan
- `askHow`
  s wants to know **r**'s applicable plan for the triggering event in the message content

Performatives in Jason

- Can think of these as achieving different aims.

Performatives in Jason

- `tell` and `untell`
  Information exchange
- `achieve` and `unachieve`
  Delegation
- `askOne` and `askAll`
  Information seeking
- `tellHow` and `untellHow`
  `askHow`
  Deliberation

Semantics

Cycle	**s** actions	**r** belief base	**r** events
1	`.send(r, tell, open(left_door))`		
2		`open(left_door)[source(s)]`	`+open(left_door)[source[s]]`
3	`.send(r, untell, open(left_door))`		
4			`-open(left_door)[source(s)]`

- Information exchange

Semantics

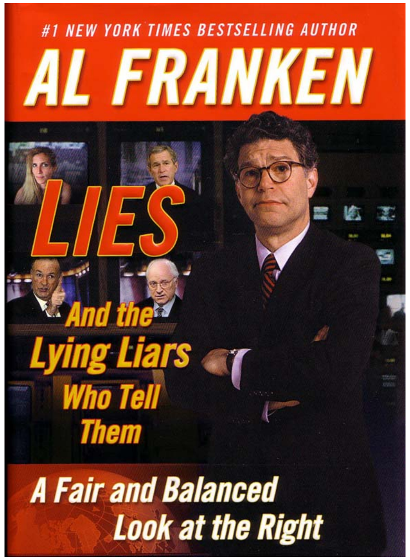
Cycle	**s** actions	**r** intentions	**r** events
1	`.send(r, achieve, open(left_door))`		
2			`+!open(left_door)[source(s)]`
3		`!open(left_door)[source[s]]`	
4	`.send(r, unachieve, open(left_door))`	`!open(left_door)[source(s)]`	
5			

- Delegation
- Note that the intention is adopted after the goal is added.

Semantics

- This semantics is operational
- Tells you how statements will be interpreted, in terms of what agents will do.
- Contrast with the mental models semantics we looked at before.

Still doesn't protect you from liars



Semantics

- `.send(receiver, tellHow, "@p ... : ... <- ...")`
  adds the plan to the plan library of **r** with its plan label `@p`

Semantics

- `.send(receiver, untellHow, PlanLabel)`
  removes the plan with the given plan label from the plan library of **r**

Semantics

- `.send(receiver, askHow, Goal addition event)`
  requires **r** to pass all relevant plans to the triggering event in the content (unlike for information seeking, this happens automatically)

Contract Net Protocol

- The CNP is a protocol for approaching distributed problem- solving
- A standardized version of the protocol has been developed by FIPA
- Agents are part of a multiagent system. They have to carry out specific tasks and they may ask other agents to perform subtasks for them
- An initiator issues a call for proposals (cfp) to all participants in the system requesting bids for performing a specific task
- After the deadline has passed, the initiator evaluates the bids it received and selects one participant to perform the task

Contract Net Protocol

- The contract net includes five stages:
  1. Recognition;
  2. Announcement;
  3. Bidding;
  4. Awarding;
  5. Expediting.

Recognition

- In this stage, an agent recognises it has a problem it wants help with.
- Agent has a goal, and either. . .
  - realises it cannot achieve the goal in isolation — does not have capability;
  - realises it would prefer not to achieve the goal in isolation (typically because of solution quality, deadline, etc)
- As a result, it needs to involve other agents.

Announcement

- In this stage, the agent with the task sends out an announcement of the task which includes a specification of the task to be achieved.
- Specification must encode:
  - description of task itself (maybe executable);
  - any constraints (e.g., deadlines, quality constraints).
  - meta-task information (e.g., "bids must be submitted by. . . ")
- The announcement is then broadcast.

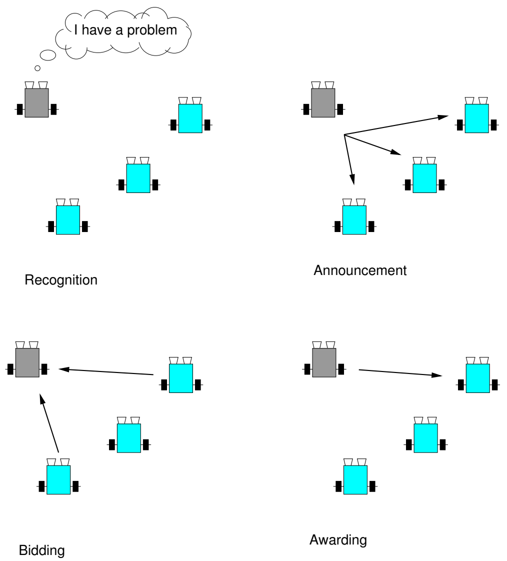
Bidding

- Agents that receive the announcement decide for themselves whether they wish to bid for the task.
- Factors:
  - agent must decide whether it is capable of expediting task;
  - agent must determine quality constraints & price information (if relevant).
- If they do choose to bid, then they submit a tender.

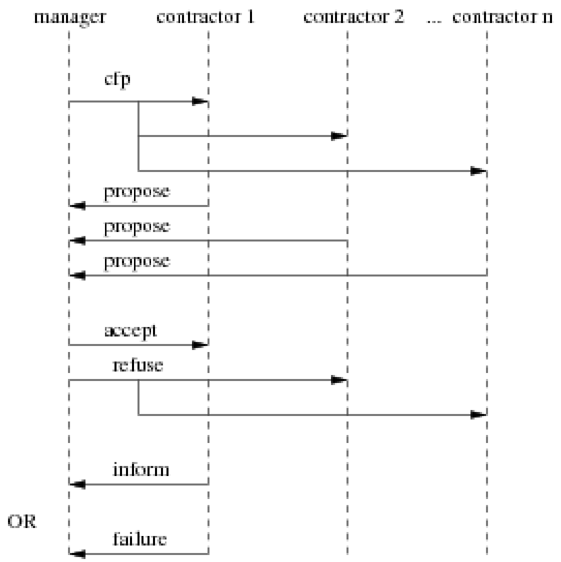
Awarding & Expediting

- Agent that sent task announcement must choose between bids & decide who to "award the contract" to.
- The result of this process is communicated to agents that submitted a bid.
- The successful contractor then expedites the task.
- May involve generating further manager-contractor relationships: sub-contracting.
  - May involve another contract net.

Stages



CNP Messages in FIPA



CNP in Jason

```
MAS cnp {

    infrastructure: Centralised

    agents:
        c;      // the CNP initiator

        p #3;   // the participants (3)
                // that offer a service

        pr;     // a participant that always
                // refuses

        pn;     // a participant that does
                // not answer
}
```

- Here's the MAS definition

An agent that doesn't respond

```
// Beliefs
plays(initiator,c).

// Plans
+plays(initiator,In)
   :  .my_name(Me)
   <- .send(In,tell,introduction(participant,Me)).

// Nothing else
```


An agent that doesn't respond

- Initial belief that c is the initiator.
- The belief that In is the initiator generates a message introducing itself.
- Nothing else.
- So, no response to any message

An agent that doesn't respond

An agent that always refuses

```
// Beliefs
plays(initiator,c).

// Plans
+plays(initiator,In)
   :  .my_name(Me)
   <- .send(In,tell,introduction(participant,Me)).

+cfp(CNPId,_Service)[source(A)] // How to respond
   :   plays(initiator,A)       // to a CfP
   <- .send(A,tell,refuse(CNPId)).
```

An agent that always refuses

- Initial belief that `c` is the initiator.
- The belief that `In` is the initiator generates a message introducing itself.
- A CfP message from an initiator will generate a `refuse` message to that agent.

Active participant

```
// Beliefs

plays(initiator,c).

price(_Service,X) :- .random(R) & X = (10*R)+100.
```

- Usual information about initiator
- `price` generates a random value for the service.

Active participant

```
// Plans

+plays(initiator,In)
   : .my_name(Me)
   <- .send(In,tell,introduction(participant,Me)).
```

- Usual response to finding out about the initiator.

Active participant

```
// Plans

@c1 +cfp(CNPId,Task)[source(A)]
   :  plays(initiator,A) & price(Task,Offer)
   <- +proposal(CNPId,Task,Offer); // remember
                                   // my proposal
      .send(A,tell,propose(CNPId,Offer)).
```

- Respond to CfP by making an offer.
- A `proposal` is added to the belief base to remember what was offered.

Active participant

```
@r1 +accept_proposal(CNPId)
   :  proposal(CNPId,Task,Offer)
   <- .print("My proposal '",Offer,"' won CNP ",CNPId,
             " for ",Task,"!").

@r2 +reject_proposal(CNPId)
   <- .print("I lost CNP ",CNPId, ".");
      -proposal(CNPId,_,_). // clear memory
```

- How to handle accept and reject messages.
- Note that there is nothing here to actually do the task.
- Refusal deletes the proposal from memory.

Initiator agent

```
// Beliefs

all_proposals_received(CNPId)
  :- .count(introduction(participant,_),NP) &
     .count(propose(CNPId,_), NO) &
     .count(refuse(CNPId), NR) &
     NP = NO + NR.

// Goals

!startCNP(1,fix(computer)).
```

- `all_proposals` counts up the proposals received.

Initiator agent

```
// Plans

// start the CNP
+!startCNP(Id,Task)
   <- .print("Waiting participants...");
      .wait(2000);  // wait participants introduction
      +cnp_state(Id,propose);   // remember the state
                                // of the CNP
      .findall(Name,introduction(participant,Name),LP);
      .print("Sending CFP to ",LP);
      .send(LP,tell,cfp(Id,Task));
      // the deadline of the CNP is now + 4 seconds, so
      // +!contract(Id) is generated at that time
      .at("now +4 seconds", { +!contract(Id) }).
```

- Send out CfP and wait for responses

Initiator agent

```
// Plans

// receive proposal
@r1 +propose(CNPId,_Offer)
   :  cnp_state(CNPId,propose)
      & all_proposals_received(CNPId)
   <- !contract(CNPId).

// receive refusals
@r2 +refuse(CNPId)
   :  cnp_state(CNPId,propose)
      & all_proposals_received(CNPId)
   <- !contract(CNPId).
```

- Here we use state information.
- If every agent has responded, then go straight to awarding the contract.

Initiator agent

```
// Needs to be atomic so as not to accept
// proposals or refusals while contracting
@lc1[atomic]
+!contract(CNPId)
   :  cnp_state(CNPId,propose)
   <- -+cnp_state(CNPId,contract);
      .findall(offer(O,A),propose(CNPId,O)[source(A)],L);
      .print("Offers are ",L);
      // must make at least one offer
      L \== [];
      // sort offers, the first is the best
      .min(L,offer(WOf,WAg));
      .print("Winner is ",WAg," with ",WOf);
      !announce_result(CNPId,L,WAg);
      -+cnp_state(CNPId,finished).
```

- Pick an offer and announce a contract

Initiator agent

```
@lc2 +!contract(_).
```

- We need a failure case — what to do if contract is called when we aren't in the proposal state.
- Why would this happen?

Initiator agent

```
// Plans

-!contract(CNPId)
   <- .print("CNP ",CNPId," has failed!").
```

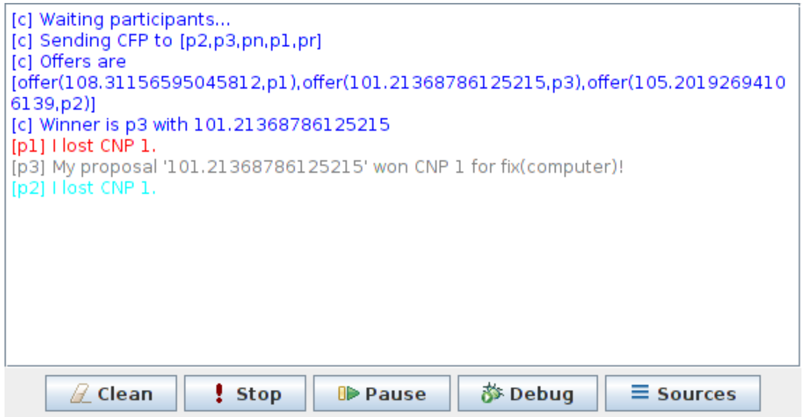
- If the `contract` goal fails for some reason.

Initiator agent

```
// Plans

+!announce_result(_,[],_).
// announce to the winner
+!announce_result(CNPId,[offer(_,WAg)|T],WAg)
   <- .send(WAg,tell,accept_proposal(CNPId));
      !announce_result(CNPId,T,WAg).
// announce to others
+!announce_result(CNPId,[offer(_,LAg)|T],WAg)
   <- .send(LAg,tell,reject_proposal(CNPId));
      !announce_result(CNPId,T,WAg).
```

- How to send out the results.
- The first clause is the base case for the recursion — do nothing.



Summary

- This lecture investigated the issue of communication in Jason, highlighting the commands for the creation of messages with different illocutionary forces.
- Some of the commands were then illustrated by discussing the code of a multiagent system implementing (a stripped down version of) the contract net protocol
- As a result, the lecture also contained a brief discussion of the contract net.