

# Robotics and Autonomous Systems

## Lecture 22: Communication in Jason

Richard Williams

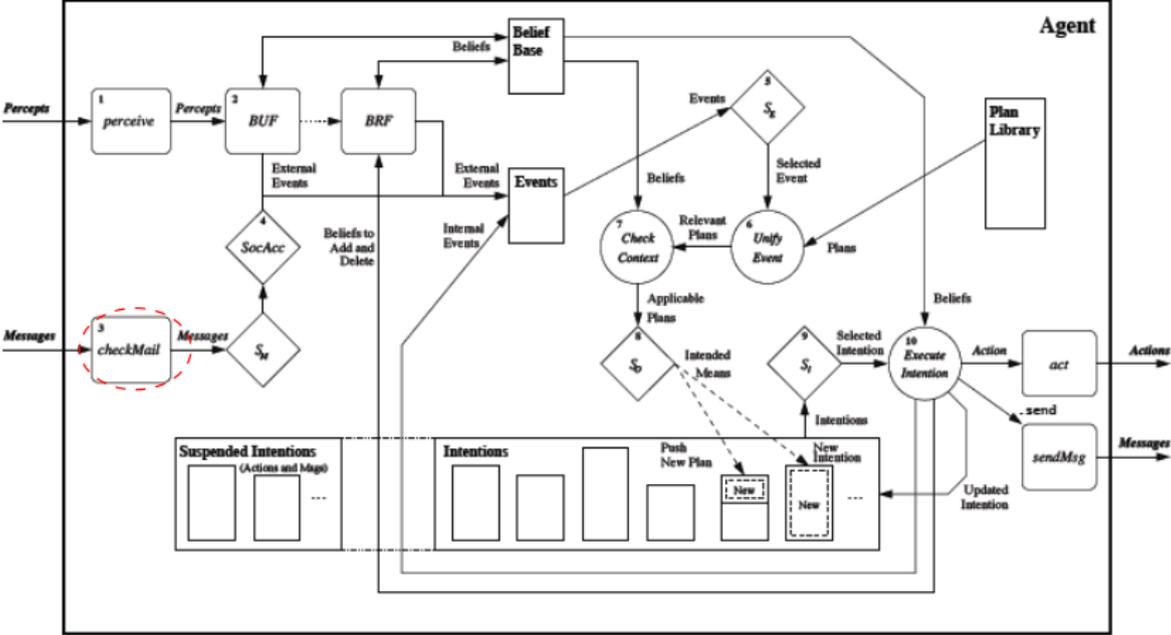
Department of Computer Science  
University of Liverpool



UNIVERSITY OF  
LIVERPOOL

- 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



- Each message received by the checkMail method (receiver's perspective) should be thought has having the form:  
`<sender, performative, content>`
- Where:
  - `sender` is the AgentSpeak term with which the agent is identified in the system
  - `performative` this represents the goal the sender intends to achieve by sending the message
  - `content` is an AgentSpeak formula (varying depending on the performative)

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

- The `.send` and `.broadcast` actions generate messages of the type `<sender, performative, 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



- 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.

- `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

- 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

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)]

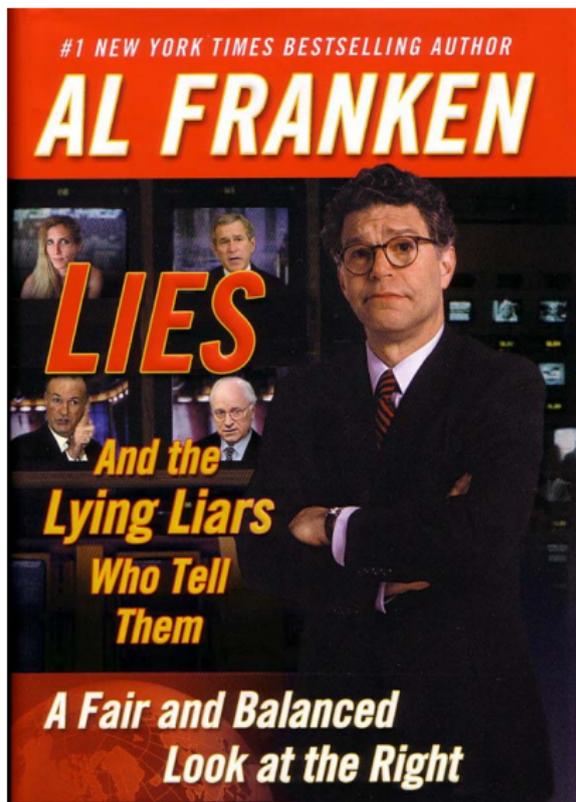
- Information exchange

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.

- 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



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

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

- `.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.

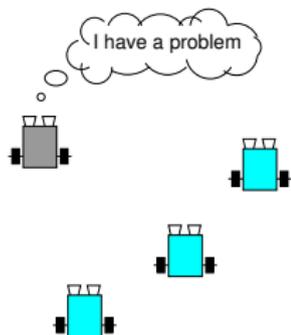
- In this stage, an agent recognises it has a problem it wants help with.
- Agent has a goal, and either. . .
  - realises it cannot achieve the goal in isolation — does not have capability;
  - realises it would prefer not to achieve the goal in isolation (typically because of solution quality, deadline,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

- 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**.

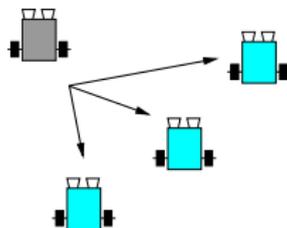
- Agents that receive the announcement decide for themselves whether they wish to **bid** for the task.
- Factors:
  - agent must decide whether it is capable of expediting task;
  - agent must determine quality constraints & price information (if relevant).
- If they do choose to bid, then they submit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

- 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.

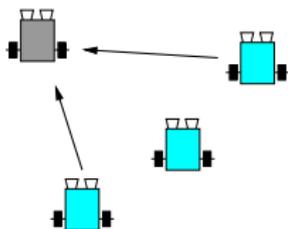
# Stages



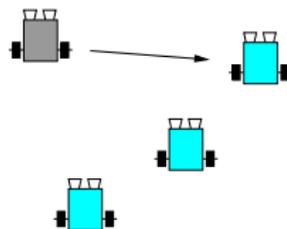
Recognition



Announcement

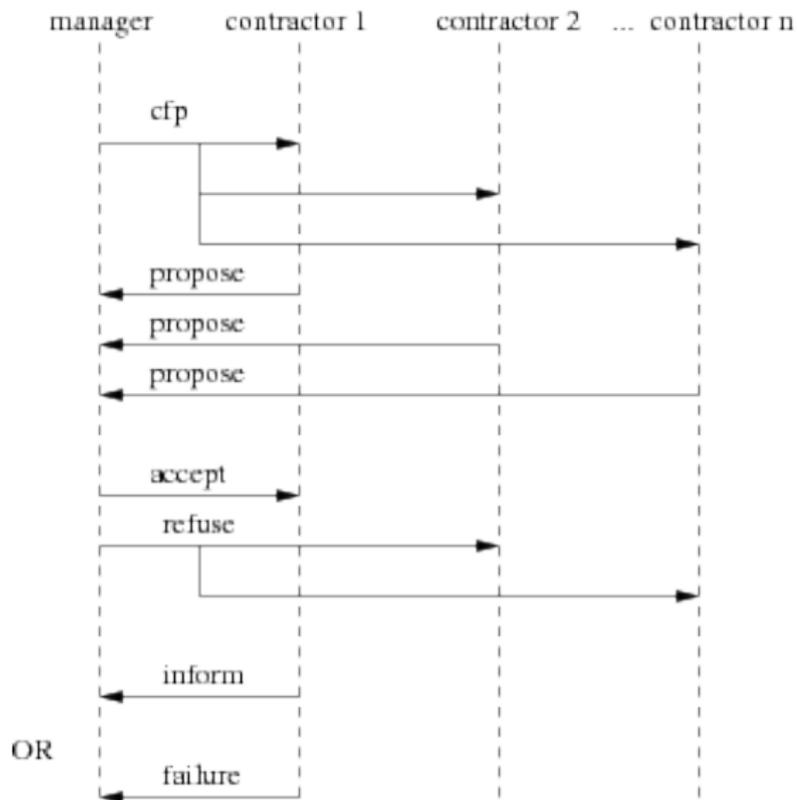


Bidding



Awarding

# CNP Messages in FIPA



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

# An agent that always refuses

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

```
// 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.

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

- Usual response to finding out about the initiator.

```
// Plans

@c1 +cfp(CNPIId,Task)[source(A)]
  : plays(initiator,A) & price(Task,Offer)
  <- +proposal(CNPIId,Task,Offer); // remember
                                     // my proposal
    .send(A,tell,propose(CNPIId,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(CNPIId)
  : proposal(CNPIId,Task,Offer)
  <- .print("My proposal '",Offer,'" won CNP ",CNPIId,
           " for ",Task,"!").

@r2 +reject_proposal(CNPIId)
  <- .print("I lost CNP ",CNPIId, ".");
     -proposal(CNPIId,-,-). // 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.

```
// Beliefs

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

```
// Goals
```

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

- all\_proposals counts up the proposals received.

```
// 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(CNPIId,_Offer)
  : cnp_state(CNPIId,propose)
    & all_proposals_received(CNPIId)
  <- !contract(CNPIId).

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

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

```
// Needs to be atomic so as not to accept
// proposals or refusals while contracting
@lc1[atomic]
+!contract(CNPIId)
  : cnp_state(CNPIId,propose)
  <- +-cnp_state(CNPIId,contract);
    .findall(offer(O,A),propose(CNPIId,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(CNPIId,L,WAg);
    +-cnp_state(CNPIId,finished).
```

- Pick an offer and announce a contract

```
@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?

```
// Plans
```

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

- If the contract goal fails for some reason.

```
// Plans

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

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

```
[c] Waiting participants...  
[c] Sending CFP to [p2,p3,pn,p1,pr]  
[c] Offers are  
[offer(108.31156595045812,p1),offer(101.21368786125215,p3),offer(105.2019269410  
6139,p2)]  
[c] Winner is p3 with 101.21368786125215  
[p1] I lost CNP 1.  
[p3] My proposal '101.21368786125215' won CNP 1 for fix(computer)!  
[p2] I lost CNP 1.
```

 Clean

 Stop

 Pause

 Debug

 Sources

- This lecture investigated the issue of communication in Jason, highlighting the commands for the creation of messages with different performatives.
- 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.