

CM30174 Coursework 2

Implementing a Supply Chain

Amit B. Kothari

ma0ak@bath.ac.uk

January 11, 2004

Abstract

This document details a solution to extend a Multi-Agent System (MAS) with an agent communication language (ACL) for a supply chain in which the (Recursive) Contract Net Protocol is used for reaching agreements. Associated code implementing *JADE* is included in the Appendix.

Contents

1	Project Definition	1
1.1	The Oracle	1
1.2	Implementation Scenario A	2
1.3	Implementation Scenario B	2
1.4	Objectives	3
2	The Contract Net Protocol	4
2.1	FIPA specification	4
2.2	Recursive Contract Net Protocol	6
3	Implementation	9
3.1	JADE	9
3.2	Programming methodologies	9
3.3	Scenario A	11
3.3.1	ClientAgent.java	11
3.3.2	SupplierAgent.java	12
3.3.3	Output Trace	13
3.4	Scenario B	15
3.4.1	Behaviour sequencing	15
3.4.2	ClientAgentB.java	16
3.4.3	Packaging subcontracts	16
3.4.4	SupplierAgentB.java	16
3.4.5	Output Trace	17
3.5	Safeguards and Inconsistencies	18
3.6	Critical Analysis and Improvements	18
A	Appendix - Scenario B conversation trace	20
B	Appendix - Simple <i>JADE</i> Agent	21
C	Appendix - Scenario A source code	22
D	Appendix - Scenario B source code	23

List of Figures

1	Recursive subcontracting	6
2	The Contract Net Protocol for Scenario A	11
3	The Contract Net Protocol for Scenario B	15

1 Project Definition

The situation is a trading scenario where client and supplier agents must be developed. A client may purchase a specified quantity of 5 items, namely *Nuts*, *Bolts*, *Washers*, *Widgets* and *Doobries* from a **single** supplier, hence the notion of a **supply chain**. The agent system consists of a number of supplier agents and a *single* client agent implemented as follows. For each run of the scenario

- Each supplier is allocated a supply of a subset of the goods at a particular cost (Determined by the *Oracle* agent - see Section 1.1). There is a fixed cost for making a whole shipment to the client, therefore shipment costs are not dependent on quantity.
- The client agent is allocated an order which must be fulfilled (Generated on each run by the *Oracle* agent).

The client agent and supplier agent should implement the Contract Net Protocol (Section 2), with the client playing the role of initiator and the suppliers playing the role of responders as follows

- The client issues a request (call for proposals or CFP) to all of the suppliers indicating the desired bundle.
- The suppliers wait for a CFP and then generate a response to the client, which may be either
 - A *Proposal*: If the supplier can satisfy the request, the supplier should issue a proposal referring to or containing the original request, and a proposed price. The price should indicate the cost of the goods, and the cost of shipping the goods to the client.
OR
 - A *Rejection*: Indicating the supplier is unable to supply the goods.
- The client waits for responses from the suppliers. If one or more proposals is received, then the client must choose a preferred proposal by cost and indicate to the winning supplier that she accepts their proposal, and to the losing suppliers that she rejects their proposals.
- The winning supplier acknowledges this to the client.

1.1 The Oracle

External information about the supply and demand for goods can be determined by a given supplier or client by interacting with the *Oracle* agent. Suppliers receive information about what they can supply by issuing a `SupplierInformationRequest` message to the *Oracle*, who will then respond with a `SupplierInformationResponse` message indicating what the supplier can supply. The client receives information about the order for a given run by issuing a `ClientOrderRequest` message to

the *Oracle*, who responds with a `ClientOrderResponse` message indicating the required quantities of particular goods desired. The *Oracle* will supply configurations of Orders and Supply information such that client requests will be satisfiable approximately 60% of the time.

1.2 Implementation Scenario A

Using *JADE*, the task is to

1. Extend the basic ontology given for the Scenario to include `AgentAction` concept classes for the message contents of
 - **The Call for Proposals**
 - **Proposals from Suppliers**

Protege and *JadeBeanGenerator* are used.
2. Provide a *JADE* agent implementation for a client and a supplier (all of the suppliers share the same implementation class) which instantiates the Contract Net Protocol (Section 2). Use the built-in Contract Net Behaviour Classes provided by *JADE* for this.

1.3 Implementation Scenario B

This is an extension to Scenario A. Whereas in the previous scenario the probability that a given request can be satisfied is relatively high, in this scenario, suppliers will be allocated supplies and orders by the *Oracle* such that the probability of a given supplier being able to completely satisfy the order on its own will be low.

Extend the supplier agents to sub-contract requests to other suppliers, in the case that they are unable to satisfy requests.

In addition to Scenario A, there will be an asymmetric fixed shipment cost between each pair of suppliers: Suppliers will be able to determine transport costs between themselves by sending a `SupplierTransportCostRequest` message containing a source supplier and a target supplier to the *Oracle*. The *Oracle* will reply with a `SupplierTransportCostResponse` message indicating the cost of making a shipment between these two suppliers.

The supplier implementation should be extended to support issuing calls and proposals to other suppliers to sub-contract the portion of the requested order that the original supplier could not meet. After a successful sub-contract is agreed the supplier should then reply to the client with a proposal in the same way as in Scenario A. Further sub-contracting in a recursive fashion may be considered, but transactions must terminate eventually.

1.4 Objectives

The problem is to implement a Multi-Agent System (MAS) model of a supply chain based on two scenarios. This breaks the immediate problem into ordered parts for Scenario A (the first target) as follows. The agents involved in this system are simple, and exhibit deductive traits. They are not strictly aligned to a subsumption architecture. However, these agents are not theorem solvers either.

1. Development of agent entities and models
2. Development of an ontology
3. Usage of a specified protocol of agent interaction
4. Building on existing code

For each part, a corresponding set of constraints and notes is derived from the problem description as follows.

1. A full implementation of the *Oracle* agent is provided. A simple **client** agent which only queries the *Oracle* for an order is provided. Finally, a simple **supplier** agent which queries the *Oracle* to determine its supplies is provided.
2. An ontology is central to each phase of the supply chain from the initial CFP to the final acknowledgement by the winning supplier. A basic ontology is provided. Each *communication* segment is viewed through time like so.
 - Client issues CFP to all suppliers. Payload - the desired bundle.
 - Each supplier generates a response to the client. Payload - {Price, Cost of Shipping} or {Rejection}.
 - For suppliers whose payload was not {Rejection}, the client generates a {Decision} to send to each of these suppliers.
 - The single winning supplier sends an {Acknowledgment} to the client.

An efficient common ontology needs to be derived for possible messages.

3. The *Oracle* generates supplier and customer configurations such that they are satisfiable approximately 60% of the time. Therefore, multiple responses of *proposals* can be expected to be returned from suppliers in response to the initial CFP by the client. The client needs the ability to rank each response and communicate its final decision to the ClientAgent.
4. *JADE* is to be used for agent classes. *Protege* and the *JadeBeanGenerator* constructs an ontology. We run this generator to produce Java source files automatically - based on the ontology project.

2 The Contract Net Protocol

The Contract Net Protocol has a FIPA standard (Section 2.1) and describes negotiation activity between a set of agents given a particular ontology. It governs messages that are expected to be sent or received in a negotiation situation of a MAS. The first part of the Protocol is a Call for Proposals (CFP) which is *generally* broadcast by the Initiator and contains a set of requirements to be fulfilled. Agents receiving this set may decide to refuse or accept this proposal, and send an appropriate response with a given cost. The Initiator then decides on the proposal with least cost and sends refusal messages to all but one of the agents that responded. The agent supplying the proposal with least cost is accepted, and the chosen agent sends an acknowledgement back to the Initiator.

2.1 FIPA specification

A Protocol has been developed and standardised by FIPA (*Foundation for Intelligent Physical Agents*). The following explanation of protocol flow derives from FIPA's standard (URL-FIPA (11/01/2003)). Document references within "[FIPA*****]", may be obtained via FIPA repositories with the given identification.

The Initiator solicits m proposals from other agents by issuing a call for proposals (cfp) act (see [FIPA00037]), which specifies the task, as well any conditions the Initiator is placing upon the execution of the task. Participants receiving the call for proposals are viewed as potential contractors and are able to generate n responses. Of these, j are proposals to perform the task, specified as propose acts (see [FIPA00037]).

The Participant's proposal includes the preconditions that the Participant is setting out for the task, which may be the price, time when the task will be done, etc. Alternatively, the $i=n-j$ Participants may refuse (see [FIPA00037]) to propose. Once the deadline passes, the Initiator evaluates the received j proposals and selects agents to perform the task; one, several or no agents may be chosen. The l agents of the selected proposal(s) will be sent an accept-proposal act (see [FIPA00037]) and the remaining k agents will receive a reject-proposal act (see [FIPA00037]). The proposals are binding on the Participant, so that once the Initiator accepts the proposal, the Participant acquires a commitment to perform the task. Once the Participant has completed the task, it sends a completion message to the Initiator in the form of an inform-done or a more explanatory version in the form of an inform-result. However, if the Participant fails to complete the task, a failure message is sent.

Note that this IP requires the Initiator to know when it has received all replies. In the case that a Participant fails to reply with either a propose or a refuse act, the Initiator may potentially be left waiting indefinitely. To guard against this, the cfp act includes a deadline by which replies should be received by the Initiator. Proposals received after the deadline are automatically rejected with the given reason that the proposal was late. The deadline is specified by the reply-by parameter in the ACL message.

Any interaction using this interaction protocol is identified by a globally unique, non-null conversation-id parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

The following items are procedures run (theoretically) on receipt of CNP messages, derived from Wooldridge (2002). A large subset of these are implemented for this project.

Task announcement processing On receipt of a task announcement, an agent decides if it is *eligible* for the task. It does this by looking at the *eligibility specification* contained in the announcement.

Bid processing Details of bids from would-be contractors are stored by (would-be) managers until some deadline is reached. A single bidder is awarded the task.

Award processing Agents that bid for a task, but fail to be awarded it, simply delete details of the task. The successful bidder must attempt to expedite the task (which may mean generating new subtasks).

Request and inform processing These messages are the simplest to handle. A request simply causes an inform message to be sent to the requestor, containing required information if it is available. An inform message causes its content to be added to the recipient's database. An inform message is also sent to the manager with the results of a concluded task.

2.2 Recursive Contract Net Protocol

Scenario B of the project demands that agents receiving a CFP from a client do not simply accept or refuse it based on their own supplies, but attempt to compensate for the *portions* of supplies that they are unable to provide. They therefore sub-contract the needs for these unmet portions and receive proposals from other suppliers. Of course, the receivers in turn may subcontract themselves, as so it goes. The *proposer* is defined as the original supplier agent needing goods. There is an optimal albeit challenging way to add this extension (Figure 1).

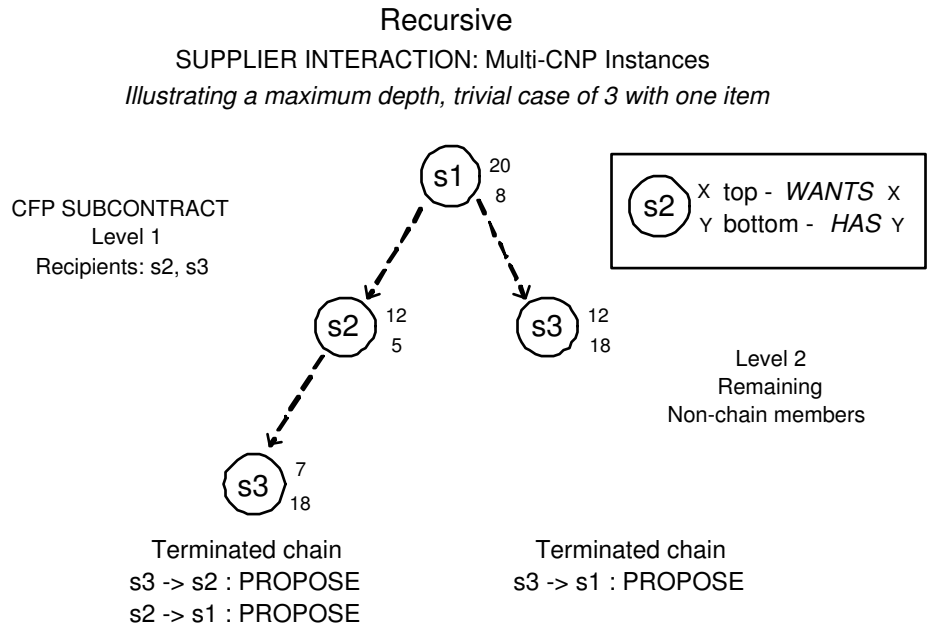


Figure 1: Recursive subcontracting

Iterated One-step subcontracting In this setting, a proposer simply chooses all suppliers in turn and asks for a proposal for the portion of goods that it is not able to fill. The target supplier sends a response which the proposer can collate and then use to determine a final response to the client. This is a simple scene where we have one round of CFP's between suppliers whose result determines the proposer's next state.

Recursive subcontracting This is a more sophisticated setting where each supplier, on receipt of a proposal from another supplier, may pass the proposal on to any other supplier not already involved if it is unable to completely fill a shortfall, best illustrated on Figure 1. We define a supplier that *received the first request* for a proposal the **originator**. This way, a chain starts from the

originator. Each agent *behind the queue* must wait until the decision tree is exhausted, possibly all the way to the leaves. A decision finally reaches the **originator**, with a crude list of chain members. The ClientAgent may then be given a report that its order will be filled by a supplier, but *using* a listed chain of other suppliers.

In **recursive subcontracting**, it is observed that we must maintain a sequential audit trail for each member agent of a particular chain. A chain begins with a proposal which goes to the **originator**. This one entry in the chain may then expand to the size of all but the original proposer. If there is n suppliers in a system, there are $n-1$ maximum entries in a chain. The set of agents which a new chain member must be picked from gets smaller with each step. However, this is for a single chain. If *one* proposer decided to try subcontracting the portion it needs to *every other* supplier, we start at maximum n chains with one entry per chain. Each chain may then explore independently, but at a smaller rate i.e. $(n-1).(n-2).(n-3)...$ and so on. Exploration is different to choice, which is made after exploration, and results in one single choice or a rejection with proposals. The resulting complexity of concurrent chains may need to be addressed carefully. As the number of agents grow, chain complexity grows exponentially. From an implementation viewpoint, usage reports suggest that an average personal computer can concurrently run 100 *JADE* agents without processing delay. However, with over 100 supplier agents, we have a potentially disastrous situation of performance with recursive subcontracting if they all have very minimal supplies. The "exploration" phase demands exponential resource to n which is typical of recursive situations.

With one-step subcontracting, many possible multi-supplier solutions (when subcontracted to a second level) would **not be found**. With recursive subcontracting, these are exploited through to the point when all possible tender agents have received tenders. Unfortunately, with supplier to supplier shipping costs, we are presented with implementation difficulties which are addressed later in this document. It is certain that implementing the recursive scenario will pose far trickier testing issues, and is initially more challenging to code.

There are shipping costs *between* suppliers. In the one-step case, one shipment cost is incurred. In the recursive case, the cost is cumulated for all chain members, and this *can* be very expensive as a final tender. The shipping arrangement through each type of subcontracting is determined as follows.

Iterated One-step subcontracting The supplier returning the least cost is chosen and ships the wanted portions directly.

Recursive subcontracting After a chain ends, it is assumed that either the chain reported a proposal from its last node (making the whole chain successful), or *exhausted* its route without success (making the whole chain a failure). Since it is possible that all the suppliers along the chain are partially supplying the goods (when successful), we must take each of their shipping

costs into account, and include them at each proposal reply from the last node, unwinding all the way to the originating node.

From the above description of **recursive subcontracting**, we can see that at every point in the chain, all other members of the supplier agents group who are not on the chain, get a CFP. This process terminates only when either there is no more suppliers left to send a CFP to, or there is a proposal from the current final node.

At each recursive step, since the CFP goes to all chain non-members, they will all reply to the CFP and the current cheapest is chosen, and this guarantees the cheapest possible **goods cost**. It is merely the shipping cost which will add overhead to an otherwise optimal strategy.

Although difficult to implement and prove irrevocably correct, the choice of **recursive subcontracting** is justified by the optimal solution it gives the ClientAgentB and the challenge it poses as a problem in this project. A solution of this kind would meet and substantially exceed the aims of Scenario B.

3 Implementation

3.1 JADE

JADE is an agent programming toolkit used to create and instantiate agent behaviour. It also has built-in behaviours that allow the Contract Net Protocol to be implemented. A typical *JADE* agent may contain code as shown in Appendix B.

JADE code is Java and requires a JRE (Java Runtime Environment) and a *JADE* installation for Windows or UNIX. See <http://jade.cselt.it> for further details. Typically, the ontology is first created by Protege's BeanGenerator plugin. Skeleton agents have been provided in this project. Certain executables also need to be included in the CLASSPATH for ease of testing. If the necessary import statements are also given in the code for the ontology package, this allows us to compile and run our agents with commands similar those given below, in one container for this project. Intra-container communication will not be considered.

```
$ javac -d [classsource] [agentsdir] [ontologydir]
$ startjade_nogui oracle:uk.ac.bath.cs.agents.OracleAgent
client:uk.ac.bath.cs.agents.ClientAgent
suppr1:uk.ac.bath.cs.agents.SupplierAgent
suppr2:uk.ac.bath.cs.agents.SupplierAgent
suppr3:uk.ac.bath.cs.agents.SupplierAgent
suppr4:uk.ac.bath.cs.agents.SupplierAgent
```

Note that to start agents for Scenario B, we must substitute `ClientAgentB` and `SupplierAgentB` into the above. We are careful to first start the *Oracle* followed by the Client and then 3 suppliers. Any number of agents can be started by naming them uniquely. Testing is carried out by observing the output from our agent code when it calls `System.out.println`. Traces of this output illustrate certain behavioural characteristics that we want to demonstrate. To implement long-term activities like a negotiation, we have to provide as many different Behaviours as there are active phases in the activity. We must also arrange for these to be created and triggered in the right sequence.

3.2 Programming methodologies

Non-functional requirements to this project include use of Object-Oriented Java with methods kept short. Further, designs are to be modular and easily extensible. The program must work correctly on the *BUCS* system in our university with the standard *JADE* toolkit. *JADE* contains Contract Net Behaviour Classes which will be used for ease of documentation and testing, namely `ContractNetInitiator` and `ContractNetResponder`. It is noted from the start that the *Oracle* agent supplied does not need development. For each scenario, SIMPLE and RECURSIVE,

there is already an appropriate data generator given the *scenario* (String) variable in the **SupplierInformationRequest** concept (for example) in the ontology. We make full use of the method handlers in the Contract Net Classes, adhering to their return value requirements and argument parameters. Sometimes (Scenario B), we over-ride these method handlers - which as the *JADE* API asserts, involves taking greater care. While reading the entirety of this section, it is advisable to consult the source code itself - available in the Appendices. The reader may find this, combined with comments in the code far more illustrative of the design decisions reached in both scenarios. Skeleton Agent code has not been documented.

3.3 Scenario A

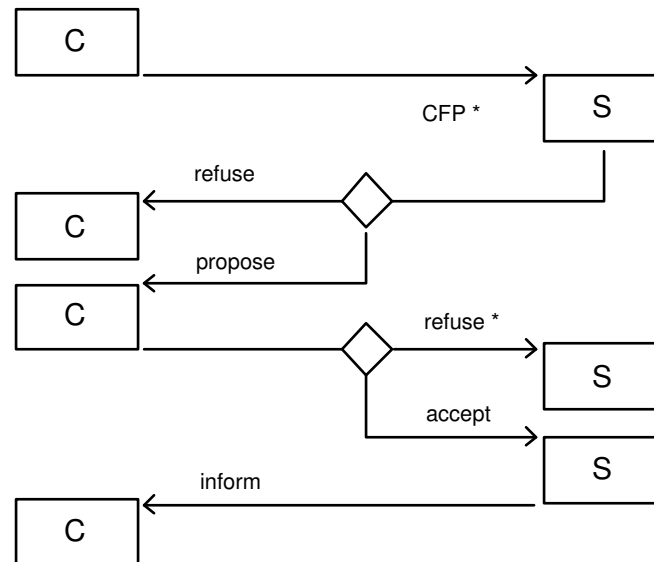


Figure 2: The Contract Net Protocol for Scenario A

This scenario implements the Contract Net Protocol described in Section 2. Figure 2 illustrates the communication processes involved in this domain. Many supporting/library methods and related code was re-used for Scenario B, and building this groundwork was a fruitful exercise.

3.3.1 ClientAgent.java

In this scenario, the Client Agent needs to do the following.

1. Get order information from the *Oracle*.
2. Create a CFP package. Broadcast this to all SupplierAgents (use DF).
3. For each response:
 - If response = rejection, ignore.
 - If response = offer, log response into datastore
4. After all responses, go through supplier offers in datastore, choose optimal package, retain SupplierID of winner.
5. Send winner SupplierID inform message, all other suppliers in datastore reject message.
6. Await inform acceptance from winner SupplierID.

A clear start to adding a suitable behaviour was extending `ContractNetInitiator`. The documentation suggests that an instance of this class is intended to be used for Contract Net Behaviour. The methods were over-written, with support/library methods added to facilitate their functionality.

DF Identification Standard strings were established to register agent services to the DF, with simple methods like `registerme()` dealing with this one-off task in `setup()`.

Supplier lookup Following on from the above, at an early stage it was realised that we could easily look up all suppliers types in the DF since they have registered themselves. The result is a standard array which is then iterated over to add recipients to the CFP in the `prepareCfps()` method of the `ContractNetInitiator`.

Behaviours For this simple case, the `ContractNetInitiator` was simply added last in the behaviour sequence of the agent.

handleAllResponses - deals with the logic of choosing the winning supplier from the proposals received and rejecting the rest. It checks first the message performative to discard refusal messages. Goods cost and shipping cost are delivered separately in the concept for clarity, and goods cost is dynamically totalled while iterating through the Vector responses.

3.3.2 SupplierAgent.java

In this scenario, the Supplier Agent needs to do the following.

1. Get supply information from *Oracle*.
2. Receive CFP package from `ClientAgent`:
 - Compare each need to own supply. If *any* not met, send rejection message.
 - If needs met, calculate proposal cost and send offer to `ClientAgent`.
3. Receive reject or accept from `ClientAgent`. If accept, send inform message to acknowledge acceptance.

To correspond with the `ContractNetInitiator` above, there is a `ContractNetResponder`, ideally suited to this Agent. We simply over-ride the default handlers in these classes to work with our desired actions. Many methods and lookups are similar to `ClientAgent.java`

Message Template To distinguish Contract Net requests, a method handler has to be passed to the `ContractNetResponder`.

Helper methods In the Responder, it was efficient to write lookups for the agent's own supplies. `getMyStock` and `getMyPrice` took the `SupplierInformationResponse` from the datastore and returned simple integers for the desired `Resource`.

Boolean ICanSupply was used to determine the final response for this simple scenario.

3.3.3 Output Trace

This section gives verbatim terminal traces from running the MAS with the *Oracle*, ClientAgent and SupplierAgents.

On this example, 4 suppliers were started. The conversation markers are shown below.

```
oracle:Starting...
Agent container Main-Container@JADE-IMTP://midge.bath.ac.uk is ready.
oracle:Registering with DF...
oracle:Starting as an oracle with seed:1073742977126
suppr3:Starting Supplier
suppr2:Starting Supplier
suppr4:Starting Supplier
client >> Starting Client
suppr1:Starting Supplier
suppr3:I have registered with the DF
suppr1:I have registered with the DF
suppr2:I have registered with the DF
suppr4:I have registered with the DF
client >> I have registered with DF
suppr3:Found my oracle called oracle
suppr1:Found my oracle called oracle
suppr2:Found my oracle called oracle
suppr3:Supply information for supplier: suppr3 with shipping to client costing 118
suppr3 1000XNut costing 52
suppr3 1000XBolt costing 19
suppr3 1000XWidget costing 48
suppr3 1000XDoobry costing 65
suppr4:Found my oracle called oracle
suppr1:Supply information for supplier: suppr1 with shipping to client costing 359
suppr1 1000XNut costing 42
suppr1 1000XBolt costing 64
suppr1 1000XWasher costing 64
suppr1 1000XWidget costing 23
suppr1 1000XDoobry costing 11
suppr2:Supply information for supplier: suppr2 with shipping to client costing 111
suppr2 1000XNut costing 80
suppr2 1000XBolt costing 70
suppr2 1000XWidget costing 73
suppr2 1000XDoobry costing 31
```

```

suppr4:Supply information for supplier: suppr4 with shipping to client costing 805
suppr4 1000XNut costing 26
suppr4 1000XBolt costing 66
suppr4 1000XWasher costing 45
suppr4 1000XWidget costing 95
client >> Supplier found called suppr4
client >> Supplier found called suppr3
client >> Supplier found called suppr1
client >> Supplier found called suppr2
client >> BEGIN. Order instance 0. I want to buy:
client >> 13XWasher
client >> 14XBolt
client >> I'm starting to prepare the CFP Jiffy bag
client >> Added receiver to CFP - suppr4
client >> Added receiver to CFP - suppr3
client >> Added receiver to CFP - suppr1
client >> Added receiver to CFP - suppr2
client >> prepareCfps() job done - CFP ready for dispatch. Vector size = 1
suppr3:prepareResponse:CFP received
suppr4:prepareResponse:CFP received
suppr1:prepareResponse:CFP received
suppr2:prepareResponse:CFP received
suppr2:prepareResponse: I have refused the CFP
suppr3:prepareResponse: I have refused the CFP
suppr1:prepareResponse: I'm tendering for this CFP. My cost: 1728 + Shipping: 359
suppr4:prepareResponse: I'm tendering for this CFP. My cost: 1509 + Shipping: 805
client >> suppr3 has not offered me a proposal
client >> suppr2 has not offered me a proposal
client >> suppr1 is quoting me a total of 2087
client >> suppr4 is quoting me a total of 2314
suppr1:MY TENDER WAS AGREED! I am sending INFORM as a job done message!
suppr4:My tender was rejected. That moron client probably thinks I'm too expensive!
client >> suppr1 informs me that my order is as good as done

```

We note that there was 2 SupplierAgents initially able to supply the order, with the most cost-effective being suppr1 by observation on the trace. Things proceeded in straightforward fashion in each phase of the Contract Net Protocol. A further run with 8 supplier agents (not given here) confirmed a working system, the verified solution to Scenario A.

3.4 Scenario B

This scenario is an extension to Scenario A, as shown in Figure 3. From a SupplierAgentB's viewpoint, on finding that it's supplies are not going to fill the order, it attempts to create a subcontract for the portion it needs. As justified previously, the challenge of **recursive subcontracting** was taken on to solve the scenario.

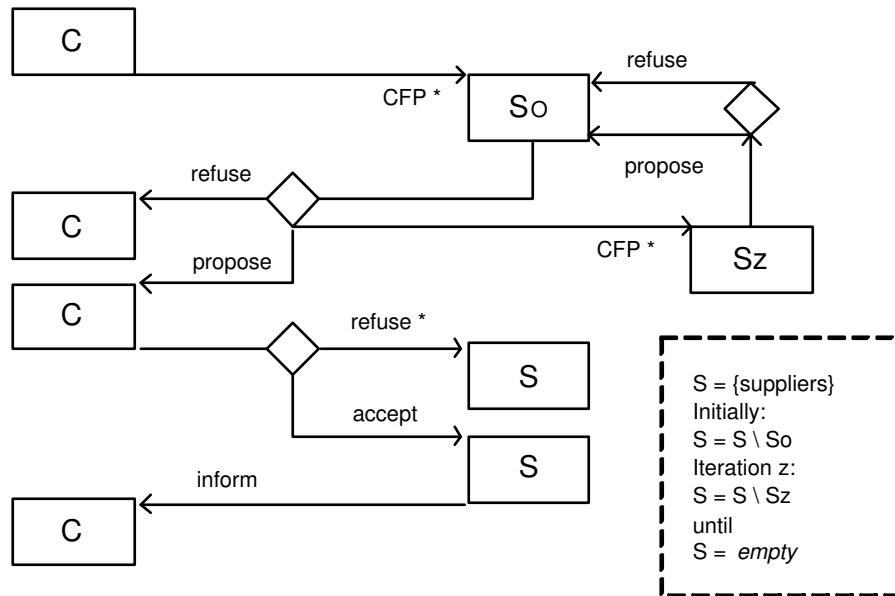


Figure 3: The Contract Net Protocol for Scenario B

3.4.1 Behaviour sequencing

Any attempt at the solution to the recursive scenario merits special attention to the correctness and termination of the parallel behaviours instantiated in SupplierAgentB. Every new ContractNetInitiator instance must remain active until resolved, passing control over to the point at which it stops once it terminates. Furthermore, the challenge is to make sure the data accumulated in this phase (PROPOSE or REJECT) with the associated ontology concept is passed to the response handler to make a decision to relay to *its* parent process. After careful consideration of the options, I realised that prepareResponse() in the ContractNetResponder was inadequate. I also wanted to avoid anything that would halt the agent and prevent its activity. After much experimentation, reading and consultation, an elegant means to achieve this was through a OneShotBehaviour that would block() until the Initiator had completed its work, over-riding the default response handler.

3.4.2 ClientAgentB.java

An important point to note is the behaviour and implementation of ClientAgentB is fairly similar as for Scenario A, since it is the suppliers that do more work in their attempt to satisfy the order. However some of the changes are explained below.

Concepts Ontology concept references were altered to cater for two upgraded concepts that are used throughout this agent as well as SupplierAgentB.

Output Output was processed in `handleAllResponses` to give an indication of the chain that was eventually passed back to us from the suppliers.

Concepts We separate two concepts which are intrinsically the same. The CFP being delivered by ClientAgentB to the SupplierAgentB uses a different Concept name to the subcontract CFP issued by the SupplierAgentB to other SupplierAgentB's.

3.4.3 Packaging subcontracts

There are five materials which a single supplier must try to supply a client. The following thoughts relate to the diversity of materials asked for in a subcontract.

One material required This is the trivial case. The subcontract has one material requested.

More than one material required If 2-5 materials are required in a given quantity, we have two options. We can package all the required materials into one subcontract and issue a proposal or we can create a separate proposal for each material required. Clearly, the proposals containing the single materials have a higher chance of receiving offers than those which have multiple materials. We need to impose a rule on all suppliers to minimise shipping cost. The first, most efficient way is to propose a subcontract for *all required materials*. If no offers are received after a deadline, the agent proposes *multiple proposals* to all other supplier agents containing a single material only.

With further consideration of the above, one realises that *none* of this is **relevant** for recursive subcontracting, since at each step a CFP goes to all remaining non-chain suppliers. They in turn, can make use of the supply they have and make up the shortfall via another CFP broadcast with a lower quantity of required goods. This note is therefore **only** applicable in the iterative one-step subcontracting case.

3.4.4 SupplierAgentB.java

A full explanation of the SupplierAgentB's role is given in the commentary contained in the code. This agent posed, by all accounts, the apex of the challenge in

this project. Finer details are given below about a mixture of methods, Behaviours and other aspects.

Datastore issues Some difficulties were faced with uses KEYS for the standard datastore. To avoid confusion in some retrievals, I used the message conversation id. Also, retrieving and storing the transport cost INFORM message (supplier to supplier) used a constant key. If many parallel behaviours are active like this, they would re-use the *current* data making shipping cost inaccurate. The aim was to resolve this without excessive code but it seemed less important than coding recursion semantics.

setup() Here we notice the wait time is less since I wanted the suppliers to be ready before the client starts up. There is an over-riding behaviour sequence of code which indicates the prepareResponse() method being taken over by a custom behaviour (for subcontracting).

Method functionality As shown, pains have been taken to neatly separate some message generating, sorting and lookup methods.

Transport Cost There were issues facing the querying of this cost and delaying progress until it was received. These have been addressed in the code and are self-explanatory in their potential for error.

Decision making The logic for preparing a response to a `SupplierProposalRequestWITHchain` includes the elimination of the simple cases of being able to supply everything and being able to supply nothing. The middle ground is the case where *some things* could be supplied, in which case we attempt **recursive subcontracting**. Being able to supply nothing that is required at all should result in a rejection since outsourcing all needs is not good practice (for shipping cost reasons also).

Objects Various problems were encountered that required various re-writes for objects like `Lists` which are interfaces, not classes.

Static vs. Non-static Static classes effectively - don't share their relation with the parent object, and are like autonomous objects. Some behaviours are static, and some where agent-scope constants needed to be accessed were left non-static with observation made about where data access might conflict with objects outside the current class.

Parallelism Clearly, an agent wants to be ready and active to consider proposals at almost any time. Attempts were made in the code to prevent completely non-responsive behaviour when waiting was required by the agent. *JADE* is helpful in that it adds behaviours in parallel by default.

3.4.5 Output Trace

The MAS was run with the *Oracle*, `ClientAgentB` and multiple `SupplierAgentB`'s. 3 suppliers were started. For the sake of brevity, the conversation trace is given in Appendix A.

3.5 Safeguards and Inconsistencies

This section describes problems encountered during development and testing.

Datastore We must ensure that keys are established for our "temporary" data which don't interfere with system constant strings, and don't overwrite current useful data at the key. In `SupplierAgentB`, I used the `conversationid` of the `ACLMessage` as a key - this can be improved.

Action/Result Wrapper Although essentially analogous to each other, these two objects should ideally be used with their implied semantic meanings. I realised their similarity and for rapid development, resorted to using Action wrappers alone.

3.6 Critical Analysis and Improvements

Deadlines were not strictly followed - there wasn't the notion of deadlines since an agent was expected to wait until all responses were received. If communication errors prevent a valid proposal reaching the originator, then we have an issue to solve. However, this is handled respectfully by various "out of sequence", "not understood" and similar error handlers. FIPA messages and string passes to methods were unchecked - and this is best done through some lookup method. A good source of reference may come from the large field of study known as **Safety Critical Systems** if correctness were to be approached properly. Any spurious information received from the *Oracle* - negative numbers, etc. was unchecked. In an ideal MAS, each agent should be insulated from spurious outside inputs triggering off their actions. If there were 2 `ClientAgents` and 2 *Oracles*, how would the behaviour be characterised? The recursive case of **multiple-client : multiple supplier** would be a natural extension to this work. Eventually one may question the very distinction between a Client and a Supplier. Aside from logistical differences, is an agent representation of one substantially different from the other for a multi-client system? The difference begins to blur when one considers allowing multiple client to broker and trade their own "wanted" orders based on prices received dynamically from the suppliers. Indeed, Clients may then decide to change their order if market conditions are not favourable for the order. The current SICS TAC competition (URL-SICS (11/01/2003)) is a real-world challenge along these lines. In Section 3.4.3, comments were made about the contents of a subcontract package. In the **iterated** case where a supplier does not meet the needs for 2-5 materials, it could address the following.

- Question whether the value of the balance of material required is more or less than the shipping cost incurred. This point is an exception - it is also applicable to the **recursive** solution.
- Instead of (a) Requesting proposals for all materials wanted (failing that)
(b) Requesting proposals for each material wanted, the agent may make additional proposals between (a) and (b), calling these *bundles*. Each proposal

bundle has more than one material in it, but not all the materials. This lessens the shipping costs if satisfied, compared to (b).

- It may be apparent that a dubiously large amount of **casting** was done of various objects especially of the Action and Result wrappers. A detailed look will make it apparent that the semantic *meaning* of these wrappers was not adhered to - in that a real result should be a **Result** and an action request should be an **Action**. This is because the concepts used within Scenario B were under the *AgentAction* abstract class in the ontology. Hence an Action wrapper had to be used. This is admittedly, not best practice.
- **IDontUnderstand** as an error catching mechanism could be improved to do context sensitive reporting on certain errors.
- Best supplier targeting - with a very large system, massive computational overhead may result (as discussed previously). We have the opportunity to “class” suppliers according to their historical probability of being able to satisfy an order. Such suppliers, that are *usually able* to satisfy order become the dependable norms or institutions. However, we should be careful to recognise that if new suppliers are introduced in future, they could market product niches in which they would become the dominant supplier in that niche. A true market emerges.

As a final note, although terminal output has been used to verify behaviour, this may not be enough. Further testing may probe each method or behaviour state to gauge the code for errors induced by input at run-time. Furthermore, the sequence of activated behaviours should be studied, and would be used to optimise SupplierAgentB. I am satisfied that both Scenarios have been rigorously addressed in this document, and represent a complete solution to the coursework.

A Appendix - Scenario B conversation trace

The independently numbered pages that follow give a terminal trace for Scenario B. Various possible supply chains can be identified for choice.

B Appendix - Simple *JADE* Agent

```
import jade.core.Agent;
import jade.core.behaviours.*;
// further imports include
// software package for system, ontology
public class SimpleAgent extends Agent
{
    protected void setup()
    {
        //ontology, etc can be set here
        //DF registration may also happen here
        addBehaviour( new myBehaviour( this ) {
            int n=0;
            public void action() {
                System.out.println( "Hello World! My name is "
                    + myAgent.getLocalName());
                n++;
            }
            public boolean done() { return n>=3; }
            //An instance of startnegotiating can be created here
            //both behaviours will run in parallel
        }
    }
    public class myBehaviour extends SimpleBehaviour
    {
        public myBehaviour(Agent a) {
            super(a);
        }
        public void action()
        { //...actions }
        private boolean finished = false;
        public boolean done() {
            return finished;
        }
    }
    //ContractNetInitiator is used for example purposes
    public class startnegotiating extends ContractNetInitiator {
        //constructor ...
        //methods like handleAllResponses and
        //prepareCfps from this class are over-written here
    }
}
```

C Appendix - Scenario A source code

Colour-highlighted source code follows this page. Both `SupplierAgent.java` and `ClientAgent.java` are given. Page numbering in this Appendix is independent of the main document.

D Appendix - Scenario B source code

Colour-highlighted source code follows this page. SupplierAgentB.java and ClientAgentB.java are included. Page numbering in this Appendix is independent of the main document.

References

URL-FIPA (11/01/2003), URL <http://www.fipa.org/specs/fipa00029/SC00029H.html>.

M. Wooldridge, *An introduction to MultiAgent Systems* (John Wiley and Sons, Chichester, 2002).

URL-SICS (11/01/2003), URL <http://www.sics.se/tac/page.php?id=13>.