CASA: HOW TO...

Rob Kremer
Department of Computer Science
University of Calgary
2500 University Dr.
Calgary, Alberta, Canada, T2N 1N4
Email: kremer@cpsc.ucalgary.ca

June 24, 2013

Abstract

CASA (Collaborative Agent Systems Architecture) is a framework for writing collaborative agents. CASA aims to support many different agent conversational paradigms, such as social commitment, BDI (Belief, Desire, Intention) and ad hoc models, and to support various standards such as the FIPA standard. But CASA also aims to avoid forcing the programmer to commit to any particular paradigm. This paper summarizes the structure of agents as well as the basic processing tasks of agents at runtime.

Contents

1 Introduction

This document shows how to do various tasks in CASA.

2 Basics

2.1 Run an agent

Agents are usually started from the command line using the `main()` method of the `casa.CASACommandLine` class or the `CASA` class. The syntax is explained in the CASACommandLine class documentation or if you run casa (through `CASACommandLine.main()`) with the qualifier `-help`. Note that a LAC agent should normally be running before you run any other CASA agent. See the CASA User Manual, Appendix E.

2.2 Write a basic agent

All you need to do to implement an agent is write your class extending casa.Agent. Make sure you override Agent's constructor with the lines:

```
public myAgent(Qualifiers quals) throws IPSocketException {
  super(quals);
}
```

You can then extend the agent as appropriate.

There are methods that you can override during an agents start-up. They all have different uses, and you should be mindful of whether the method is executing in the agent's thread or the thread of the caller of the agent's constructor.

If you need to execute code during the agent's idle time (the time when the agent isn't handling incoming messages etc), you need only override the agent's doIdle() method:

```
@Override
protected boolean doIdle () {
  boolean superRet = super.doIdle();
  boolean ret = code;
  return superRet || ret;
}
```

Don't forget to call the super version of the method. You should return true iff either the super implementation returns true or your code does something. You should avoid doing anything too lengthy during the idle call, and return control to the outer processing loop within a second or two. You may need to divide up your work: do some of it now, return, and carry on next time the `doIde()` method is called.

Warning: `doIdle()` is called whenever an agent becomes idle (ie: many times), so any code here will be run many times. You have to be smart about having the code run selectively.

2.3 Deferring execution of code until later

There are situations where you want to execute code later or the current threat is not the agent's "normal" thread (eg: the AWT thread) and the code must be executed in the agent's main thread. To do

Method	Thread	Description
Constructor	caller/creator	Be sure to call the parent constructor first in the constructor. Since the agent isn't initialized at all here, there is not likely much you'd want to do in the constructor.
initializeConstructor (AbstractProcess.Qualifiers)	caller/creator	called just before the message loop begins; the agent is not fully initialized at this point – it's not yet registered with the LAC. You can override this method to deal with qualifiers specific to your agent. See @link casa.AbstractProcess#initializeRun(Qualifiers) and the top-level documentation of @link casa.AbstractProcess for more detail.
initializeThread()	agent	override this (but be sure to call the super method) to deal with an initialization, but the agent is not yet registered with the LAC, and will not yet have recovered any of it's persistent data (if it is a persistent agent).
initializeAfterRegistration()	agent	called just after the agent is registered with the LAC – the agent is now fully initialized.
pendingFinishRun()	agent	called when an exit() has been called while the agent is still running. Called only once.
finishRun()	agent	called after the message loop has exited (when the agent has just stopped running). Called only once Execute code during idle time

Table 1: default

this, just wrap the code in a Runnable object, and call the method `AbstractProcess.defer(Runnable)` or `AbstractProcess.defer(Runnable, long)` where the first parameter is a `Runnable` object (typically an anonymous class type) and the second parameter is represents the minimum delay time in milliseconds. For example, the following code embedded in an agent, will print "Hello world" to standard out one second or more after it is executed in the thread of the agent:

```
...
Runnable runnable = new Runnable(){public void run(){
  println("Hello world");
  }};
defer(runnable, 1000);
...
```

3 Message Handling

3.1 Handle an incoming message

If you really need to handle a message directly, you can override `TransientAgent.handleMessage(MLMessage)`, but this is ***not*** recommended. If you do it, be sure to call the superclass's version if you don't handle the message.

3.2 Handle a new type of request

There is social commitment *request* conversation skeleton defined as a Lisp function. Therefore, to instantiate a new type of request, you need only to execute that function, and define (at least) the handler for the servicing or the request (in the serving agent) and a handler for dealing with the result (in the client agent). The best place to do this in your `<angent-name-or-agent-type-name>.init.lisp` file, which while be executed whenever your agent starts up (see CASA User Manual, Section 4.2).

For example, the `TransientAgent` class defines an "execute" request that requests another agent to execute some Lisp command on its behalf, and send back the result. In this case, `TransientAgent` is *both* the client and server of this conversation type, so it defines in its `casa.TransientAgent.init.lisp` file[1] the following:

```
(request-server "execute"
  `(jcall
     (jmethod (agent.get-class-name) "perform_execute" "casa.MLMessage")
     agent (event.get-msg)
     )
  )

(request-client "execute"
  `(jcall
     (jmethod (agent.get-class-name) "release_execute" "casa.MLMessage")
     agent (event.get-msg)
     )
  )
```

[1] This file needs to be placed in any one of the places defined in the CASA User Manual, Appendix A.

These functions are called with minimum arguments, that only define the name of the *act* of the request message, and task to be done when the server provides the server, or the client handles the answer back, respectively. In both cases, these are handled by Java method calls. If desired, you can specify further detail using various optional keyword parameters[2].

The server specifies it's method for actually doing the service:

```
public PerformDescriptor perform_execute (MLMessage message) {
  String content = message.getParameter(ML.CONTENT);

  // if the content is quoted, strip the quotes
  while (content!=null && content.length()>0 && content.charAt(0)=='"'
                                  && content.charAt(content.length()-1)=='"') {
    try {
      content = CASAUtil.fromQuotedString(content);
    } catch (ParseException e) {
      return new PerformDescriptor(new Status(-1,"Malformed content field: \""
                 +content+"\".  Expected a legal run-time command"));
    }
  }

  PerformDescriptor ret = new PerformDescriptor();
  try {
    BufferedAgentUI ui = new BufferedAgentUI();
    Status execResult = abclEval(content, null, ui);

    //send back FAILURE, SUCCESS or PROPOSE depending on a status value of <0, 0 or >0.
    if (execResult.getStatusValue()<0) { //FAILURE return
      ret.put(ML.PERFORMATIVE, ML.FAILURE);
      ret.put(ML.CONTENT, CASAUtil.serialize(content, execResult, ui.result()));
      ret.put(ML.LANGUAGE,"casa.*");
    }
    else { // successful PROPOSE return
       ret.put(ML.PERFORMATIVE, ML.PROPOSE);
       ret.put(ML.CONTENT, CASAUtil.serialize(content, execResult, ui.result()));
       ret.put(ML.LANGUAGE,"casa.*");
    }
  } catch (Throwable e) {
     ret = new PerformDescriptor(-4,println("error","Failed to execute command from agent ``+
                       ''message.getParameter(ML.SENDER)+": "+content,e));
     ret.put(ML.PERFORMATIVE, ML.FAILURE);
     ret.put(ML.CONTENT, CASAUtil.serialize(ret));
     ret.put(ML.LANGUAGE,"casa.*");
   }
  return ret;
}
```

[2]For example, the server may want to specify a decision process `agree` or `refuse` to do the task.

Fundamentally, this method receives the actual request message object as a parameter, attempts to execute the lisp code in the content of the message, and then only needs to return a `PerformDescriptor` (which not much more than a simple dictionary) with the non-default keys filled in, which the system will use to construct a reply to the message. The reply includes the result of the Lisp execution in the `:content` field.

The client's code, which is called upon receiving the reply is similar:

```
public PerformDescriptor release_execute (MLMessage message) {
  String lang = message.getParameter(ML.LANGUAGE);
  String performative = message.getParameter(ML.PERFORMATIVE);
  String content = message.getParameter(ML.CONTENT);

  String problem = "array";
  try {
    Object o = CASAUtil.unserializeArray(content);
    Object[] objects = (Object[])o;
    problem = "originalCommand String";
    String originalCommand = (String)objects[0];
    problem = "Status object";
    Status stat = (Status)objects[1];
    problem = "text String";
    String text = (String)objects[2];
    problem = "object from Status";
    Object object = (stat instanceof StatusObject<?>)
              ? ((StatusObject<?>)stat).getObject()
              : null;

    //boolean html = (commandInterpreter.getStyle()==RTCommandInterpreter.STYLE_HTML);
    boolean html = false;

    if (stat!= null && stat.getStatusValue () >= 0) {
      notifyObservers (ML.EVENT_POST_STRING, "\n"
           +(html?"<font color=blue><b>":"")
           +"Command result from "+message.getFromString()+":"
           +(html?"</b>":"")
           +"\n"+stat
           +"\n"+text
           +(html?"</font>\n":"\n"));
    } else {
      if (isLoggingTag("warning")) println ("warning",
                  "TransientAgent.release_execute: got bad status in message:\n"
                  + message.toString (true));
      notifyObservers (ML.EVENT_POST_STRING, "\n+"
         +(html?"<font color=red><b>":"")
         +"Failed command result from "+message.getFromString()+":"
         +(html?"</b>":"")
         +"\n"+stat
```

```
          +"\n"+text
          +(html?"</font>\n":"\n"));
      }
      return new PerformDescriptor();
    } catch (Throwable e) {
    PerformDescriptor ret = new PerformDescriptor(new Status(-747,
               println("error",
               "TransientAgent.release_execute: cannot unmarshal '"+problem
               +"' from CONTENT field \n  \""+content
               +"\"\n  -- expecting a tuple "
               +"[originalCommand:String status:Status textOutput:String]"
               +problem,e)));
    ret.put(ML.LANGUAGE, "text");
    return ret;
    }
  }
```

The client receives the `propose/discharge` message from the server as it's parameter, parses out the reply, returns a `PerformDescriptor` specifying the acceptability or otherwise of the result (which would release the server of the social commitment to perform the task or not, respectively). Notice that in the case of success, the client need not write *anything* into the PerformDescriptor, but just leaves it as default, which the system automatically takes as success, and replies affirmatively. The client could have returned `null` with the same affect. Likewise, the failure conditions also minimize the information packed into the `PerformDescriptor` as a negative status integer indicates to the system that a `refuse/propose|discharge` should be returned.

If you need more detailed control over the various actions at parts of the conversation, the server function is defined as follows:

```
(defun request-server
  ( ;will be named: the-act+"-request-server"
    the-act
    result-action; when the server responds with propose/discharge... this executed in the context of tha
    &key
    (base-name "request-server")
    (name (concatenate 'string the-act "-" base-name))
           ;the actual request-like performative, override only to change the performative
    (request-performative request)
    ;setting this gives the agent a chance to AGREE of REFUSE
    (request-decision `(performdescriptor 0 :performative agree))
    ;the action to perform when we receive a AGREE reply to our PROPOSE
    agree-action
    ;the action to perform when we receive a REFUSE reply to our PROPOSE
    (refuse-action `(exception-handler ,name "received unexpected refuse"))
    ;the action to perform when we receive a NOT_UNDERSTOOD reply to our PROPOSE
    (not-understood-action `(exception-handler ,name "received unexpected not-understood"))
    (timeout-action `(exception-handler ,name "received unexpected timeout"))
    agree-discharge-action
    (refuse-discharge-action `(exception-handler ,name "received unexpected refuse-discharge"))
```

```
    (not-understood-discharge-action '(exception-handler ,name "received unexpected not-understood-discha
    (timeout-discharge-action '(exception-handler ,name "received unexpected timeout-discharge"))
    nopropose
    transformation
    optional-negotiation
    ;the action to send a propose/discharge message containing the result of performing the perform-actio
    &aux
    (ask-name (concatenate 'string name "-ask"))
    (offer-name (concatenate 'string name "-offer"))
    (approver-name (concatenate 'string name "-approver"))
    )
    ...
)
```

For the client side:

```
(defun request-client
  ( ;will be named: the-act+"-request-client"
    the-act
    result-action; when the server responds with propose/discharge... this executed in the context of tha
    &key
    (request-performative request)
    (request-act the-act)
    (base-name "request-client")
    (name (concatenate 'string the-act "-" base-name))
    ;;for the ask-client conversation
    ;setting this gives the agent a chance to AGREE or REFUSE a proposal from from the server; default: r
    (propose-decision '(performdescriptor 0 :performative refuse))
    ;the action to perform when we receive a AGREE reply to our REQUEST
    agree-action
    ;the action to perform when we receive a REFUSE reply to our REQUEST
    (refuse-action '(exception-handler ,name "received unexpected refuse"))
    ;the action to perform when we receive a NOT_UNDERSTOOD reply to our REQUEST
    (not-understood-action '(exception-handler ,name "received unexpected not-understood"))
    (timeout-action '(exception-handler ,name "received unexpected timeout"))
    ;;for the discharge-client conversation
    (failure-action '(exception-handler ,name "received unexpected failure"))
    (timeout-discharge-action '(exception-handler ,name "received unexpected timeout-discharge"))
    nopropose
    transformation
    optional-negotiation
    ;the action to send a propose/discharge message containing the result of performing the perform-actio
    &aux
    (ask-name (concatenate 'string name "-ask"))
    (offer-name (concatenate 'string name "-offer"))
    (approver-name (concatenate 'string name "-approver"))
    )
```

```
  "Creates a Conversation for a client side request, supporting:
  1. an outgoing REQUEST while in state not-started (->started)
  and incoming messages AGREE, REFUSE or not-understood when in state started (->terminated); or
  2. an incoming PROPOSE while in state not-started (->started)
  and an outgoing AGREE or REFUSE when in state started (->terminated)"
    ...
)
```

3.3 Deferring processing a "callback" from an incoming message

Assuming you're using "callbacks" in a conversation to handle messages, you will occasionally run into the situation where you have to wait for something else to happen (eg: a user making a decision in a dialog box; a conversation with another agent to terminate; etc.). The easiest way to handle this is to just let your callback be called, and in it's body, check on the event you are waiting on. If it's happened, proceed; but it it hasn't yet happened just return the value of `DEFER_ACTION` (a constant defined in `TransientAgent`). Since all "callbacks" return a `PerformDescriptor`, your return call will be:

```
new PerformDescriptor(new Status(DEFER_ACTION));
```

This return will cause the system to call your "callback" again at a later time. You can repeat returning `DEFER_ACTION` as many times as you want.

3.4 Ignore an incoming request (and not reply)

From a method answering called in response to determine a *reply* message in a *request* conversation, you can just not reply by returning new `PerformDescriptor(new Status(DROP_ACTION))`. In the social commitment communication paradigm, this will discard the associated commitment to reply only if the originating request was a broadcasted message (`MLMessage.isBroadcaste()`).

3.5 Send a message to another agent

Call one of the following methods:

```
AbstractProcess.sendMessage(MLMessage) : Status
TransientAgent.sendMessage(String, String, URLDescriptor, String...) : Status
```

The second one calls the first, but constructions a message for you out of the parameters (the list is an array of key/value pairs (keys are even, values are odd). Key may not be null, but values may be.

```
AbstractProcess.sendRequestAndWait(
      MLMessage message,
      long timeout,
      MessageEventDescriptor... messageDescriptors) : StatusObject<MLMessage>
```

This methods sends the message and waits for a response message that matches any one of the messageDescriptors and returns that message. If none of the messageDescriptors are matched within timeout milliseconds of the current time, the method returns a negative status and a null in place of the MLMessage. If you omit the `MessageEventDescriptors` altogether, they will default to those appropriate for a *request* – `agree, refuse, not-understood` and `failure`.

4 Making an agent and it's attributes persistent

An agent is persistent if it inherits from the `Agent` class and sets the attribute persistent to true (which can be done either in the agent code or via the command line). The agent stores the information in a file according the the `LAC`'s setup. All you have to do to make at attribute persistent is to mark it's declaration with the `@Persistent` annotation. For example:

```
/** a simple persistent boolean attribute to be stored in the properties under "myFlag" */
@Persistent
boolean myFlag;

/**
 * a persistent object that will be stored in the properties under "options.x"
 * and "options.y" because it has at least one @Persistent attribute itself.
 */
public class Options {
  @Persistent int x = 4;
  @Persistent double y = 7.5;
}

@Persistent
Options options;

/**
 * a persistent object that will be stored in the properties under "stuff" in
 * the standard CASA serial format (because none of it's properties are marked
 * @Persistent).  Note that the class Stuff MUST have a toString()
 * method and a corresponding constructor that takes a single string.
 */
public class Stuff {
  int x = 4;
  double y = 7.5;
  public Stuff() {...}
  public Stuff(String persistData) {...}
  @Override
  public toString() {...}
}

@Persistent
Stuff stuff = new Stuff();
```

5 Cooperation Domains

5.1 Send a request through a cooperation domain

This section gives an example of how to carry on a *request* conversation between two or more agents who are members of a cooperation domain using the CD as an intermediary for the conversation.

To send create and send the initial `request` message. This is done by constructing a typical request message and then using `CooperationDomain.constructCDProxy()` to stuff the request in into a proxy message before sending it. The form of `CooperationDomain.constructCDProxy()` shown here will construct a broadcast message (signified by a "*" for the final receiver). You can exercise more control over the destination address by using the two additional polymorphic forms of `CooperationDomain.constructCDProxy()`.

```
MLMessage msg = MLMessage.getNewMLMessage().setParameters(new String[]{
     ML.PERFORMATIVE,ML.REQUEST,
     ML.ACT,          "myAct",
     ML.CONTENT,      "myContent"
     ML.REPLY_WITH    "A UNIQUE STRING" // <- this is important - the system won't
                                        //match the reply with the request without it
     });

//Since we're going through a CD, we have 3 options for how we want the the server
//to reply: we can ask it to reply directly to us, reply indirectly through the
//CD, or indirectly through the CD but also let everyone else "hear".
switch (replyMethod) {
case direct: // have the server reply directly to me, bypassing the CD
  //explicitly set the REPLY_TO to be direct
  msg.setParameter(ML.REPLY_TO, getURL().toString());
  break;
case publicThruCD: //have the server reply through the CD in directed mode (everyone hears)
  cd.setDataValue("directed", null); //This tells the CD to use directed mode
  break;
case thruCD: //have the server reply through the CD (in whisper mode by default)
  //this is the default behaviour, so no need to do anything special
  break;
}

MLMessage proxy = CooperationDomain.constructCDProxyMessage(msg, getURL(), cd);
sendMessage(proxy);
```

There's not a lot of difference between an ordinary consider() and one recieving a broadcaste, but the difference is important. This example is for a shortcutting server. Most of the code here is directly from the template for a shortcutting request server (see above), but also the code in red is used to ensure that the client is forced to reply back through through the CD. You need not have the client reply through the CD (and have the client reply directly back) if you want. You could also force the client to reply back in "directed mode" (everyone can "hear") by adding a line "sender.setDataValue("directed", null)" and adding the sender in ML.RECEIVER in the return PerformDescriptor (as we did in the previous section).

```
public PerformDescriptor consider_myAct(MLMessage msg)
```

```
URLDescriptor sender=null;
try
  sender = new URLDescriptor(msg.getParameter(ML.SENDER));
 catch (URLDescriptorException e)
  e.printStackTrace();

PerformDescriptor ret = new PerformDescriptor();

ret.put(ML.PERFORMATIVE, ML.SUCCESS); //shortcutting
ret.put(ML.ACT, msg.getAct().push(ML.DISCHARGE).toString());
// if we received this msg through a CD, we want to force the client to reply to us
// through the CD too.
if (msg.getParameter(ML.CD)!=null)
  URLDescriptor replyto = getURL();
  replyto.pushViaAtEnd(sender); //can use sender or cd -- hopefully they're the same.
  ret.put(ML.REPLY_TO, replyto.toString());


//TODO compute the result and put it in the CONTENT of the return message
ret.put(ML.CONTENT, "myContent");
return ret;
```

The client code for `release_myAct()` should be similar code to the coloured code above in order to properly direct any replies. The same applies to any code that leads to the generation of messages.

5.2 Paying attention to (or ignoring) messages arriving from CDs not addressed to you

Messages sent through a CD in "directed mode" (so everyone in the CD can "hear"), will arrive in every member agent's message inbox even though they are not addressed to that agent. By default CASA will merely dropped these messages since they are not addressed to you. But you can change that be calling setObserveMessages(true), which will allow processing of the incoming messages and call handler methods (consider_*()-type methods) in your agent. The methods called are similar to the "normal" ones but with "_evesdrop" appended to them. Eg: instead of conclude_act(MLMessage), it's conclude_act_evesdrop(MLMessage). If the appropriate evesdrop method isn't found, TransientAgent.evesdrop(MLMessage) will be called (which does nothing be return DROP_ACTION); your agent can override this to exhibit whatever behaviour you'd like.

6 Interfaces

6.1 Add a runtime command to an agent

The runtime language of CASA is Lisp, so to add a runtime command, you define a new Lisp command, which can easily be done in Java. For details see the CASA User Manual, Section 11.3)

6.2 Add a menu or menu item to an agent's default graphical interface

Subclass the appropriate subclass of `AbstractInternalFrame`, override `makeMenuBar()`:

```
protected JMenuBar makeMenuBar () {
  JMenuBar menuBar = super.makeMenuBar();
  // manipulate menuBar here.
}
```

There are several helpful mentors in the `AbstractInternalFrame` class that help with manipulating the menus and menu items:

```
public void insertMenuBar(JMenu menu, int location)
```

Inserts a new menu in the agent menu bar at the specified location. The remaining items are moved over one.

```
public void insertMenuBarAfter(JMenu menu, String name)
```

Inserts a new menu in the agent menu bar after the named menu label.

```
public void insertMenuBarBefore(JMenu menu, String name)
```

Inserts a new menu in the agent menu bar before the named menu label.

```
public void replaceMenuBar(JMenu menu)
```

Replace the menu with the same name. If the name doesn't exist, the new menu is added at the end.

```
public JMenu getMenuBarMenu(String name)
```

Returns the menu with the label `name`. You can then modify it as appropriate and reinsert it using `replaceMenuBar()`.

6.3 Add a tab pane to an agent's default graphical interface

Subclass the appropriate subclass of `AbstractInternalFrame`, override `makeMenuBar()`:

```
protected JMenuBar makeMenuBar () {
  JMenuBar menuBar = super.makeMenuBar();
  // manipulate menuBar here.
}
```

```
public void putTab(String title, Component component)
```

Add a new tab a new title and component to the tab pane of the interface.

```
public Component getTab(String name) {
```

Returns the component associated with the tab with the named label. If no such label is found, returns null.

```
public void removeTab(Component component)
```

Remove the tab with the specified component from the tab pane of the interface.

```
public void removeTab(String name)
```

Remove the tab with the specified label from tab pane of the interface.

```
public boolean setSelectedTab(String name)
```

Sets the tab with the specified name as the new currently selected tab.

6.4 Add a new interface to an agent

The type of the interface is dictated by whether or not a graphic environment is present. So there are two different methods you might override. Code in `TransientAgent` will decide which to call. Both return an instance of `casa.ui.AgentUI` or null (if no interface is desired). Note that these methods will only be called if a specific interface isn't specified on the command line.

```
TransientAgent.makeDefaultTextInterface(String[], String)
TransientAgent.makeDefaultGUIInterface(String[], String)
```

Acknowledgments

The author wishes to acknowledge the Canadian National Science and Engineering Research Council (NSERC) for financial support of the research.